

conference
.....
proceedings

2009 USENIX Annual Technical Conference

*San Diego, CA, USA
June 14–19, 2009*

Sponsored by
The USENIX Association
USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION



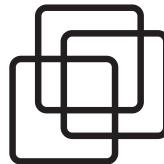
Proceedings of the 2009 USENIX Annual Technical Conference

San Diego, CA, USA June 14–19, 2009

ISBN: 978-1-931971-68-3

Thanks to Our Sponsors

Silver Sponsor



vmware®

Bronze Sponsors

Google™



Microsoft®

Research



Sponsor
BERKELEY
COMMUNICATIONS
Your Storage & Networking Partner

Media Sponsors

Hackett and Bankwell:
The Linux Comic

Help Net Security

IEEE Security & Privacy

InfoSec News

Linux Gazette

Linux Journal

Linux Pro Magazine

LXer.com
Messaging News
Network World ITRoadmap
No Starch Press
SNIA
Toolbox.com
UserFriendly.org

ACM Queue
Addison-Wesley Professional/
Prentice Hall Professional/
Cisco Press
BetaNews
ConferenceGuru
Distributed Management
Task Force, Inc.
Free Software Magazine

© 2009 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN: 978-1-931971-68-3

The USENIX Association

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

For more information about membership and its benefits, conferences, or publications, see <http://www.usenix.org>.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at <http://www.sage.org>.

Thanks to USENIX & SAGE Corporate Supporters

USENIX Patron

Microsoft®
Research

USENIX Benefactors

hp
invent

Infosys
POWERED BY INTELLECT
DRIVEN BY VALUES

LINUX
MAGAZINE

NetApp™

Sun
microsystems
The Network is the Computer™

vmware®

USENIX & SAGE Partners

Ajava Systems, Inc.
DigiCert® SSL Certification
FOTO SEARCH Stock Footage
and Stock Photography
Hyperic Systems Monitoring
Splunk
Zenoss

USENIX Partners

Cambridge Computer Services, Inc.
GroundWork
Open Source Solutions
Xirrus

SAGE Partner

MSB Associates

USENIX Association

**Proceedings of the
2009 USENIX Annual Technical Conference**

**June 14–19, 2009
San Diego, CA, USA**

Conference Organizers

Program Co-Chairs

Geoffrey M. Voelker, *University of California, San Diego*
Alec Wolman, *Microsoft Research*

Program Committee

Remzi Arpacı-Dusseau, *University of Wisconsin, Madison*
Ranjita Bhagwan, *Microsoft Research India*
George Canea, *EPFL*
Ira Cohen, *HP Labs, Israel*
Landon Cox, *Duke University*
John Dunagan, *Microsoft Research*
Nick Feamster, *Georgia Institute of Technology*
Michael J. Freedman, *Princeton University*
Garth Goodson, *NetApp*
Robert Grimm, *New York University*
Dirk Grunwald, *University of Colorado*
Jaeyeon Jung, *Intel Research*
Sam King, *University of Illinois at Urbana-Champaign*
Geoff Kuenning, *Harvey Mudd College*
Ed Lazowska, *University of Washington*

Erich Nahum, *IBM T.J. Watson Research Center*

Niels Provos, *Google, Inc.*

Sean Rhea, *Meraki, Inc.*

Mahadev Satyanarayanan, *Carnegie Mellon University*

Prashant Shenoy, *University of Massachusetts*

Marvin Theimer, *Amazon.com*

Andrew Warfield, *University of British Columbia and Citrix Systems*

Yinglian Xie, *Microsoft Research Silicon Valley*

Ken Yocum, *University of California, San Diego*

Poster Session Co-Chairs

George Canea, *EPFL*

Andrew Warfield, *University of British Columbia and Citrix Systems*

Invited Talks Program Committee

Dan Klein, *USENIX Association*

Ellie Young, *USENIX Association*

The USENIX Association Staff

External Reviewers

Atul Adya	Cristian Estan	Jie Liu	Hovav Shacham
Bhavish Aggarwal	Steve Gribble	Susan Martonosi	Neil Spring
Muneeb Ali	Chris Grier	Trevor Pering	Kiran Srinivasan
David Becker	Dan Halperin	Alkis Polyzotis	Radu Stoica
Sapan Bhatia	Michael Hicks	Vijayan Prabhakaran	John Strunk
Nilton Bila	Wenjun Hu	Moheeb Rajab	Sai Susarla
Silas Boyd-Wickizer	Hai Huang	Charlie Reis	Srinivasan Venkatachary
Luis Ceze	Rahul Iyer	Eric Rescorla	Michael Vrable
Vitaly Chipounov	Horatiu Jula	Yaoping Ruan	Yi Wang
Anthony Cozzie	Charles Krasic	Stefan Saroiu	John Zahorjan
Tim Deegan	Arvind Krishnamurthy	Jiri Schindler	Cristian Zamfir
Fred Douglis	Geoffrey Lefebvre	Simon Schubert	Nickolai Zeldovich
Jeremy Elson	James Lentini	Vyas Sekar	Lintao Zhang

2009 USENIX Annual Technical Conference

June 14–19, 2009

San Diego, CA, USA

Message from the Program Co-Chairs vii

Wednesday, June 17

Virtualization

Satori: Enlightened Page Sharing	1
<i>Grzegorz Miloś, Derek G. Murray, and Steven Hand, University of Cambridge Computer Laboratory; Michael A. Fetterman, NVIDIA Corporation</i>	
vNUMA: A Virtual Shared-Memory Multiprocessor	15
<i>Matthew Chapman, The University of New South Wales and NICTA; Gernot Heiser, The University of New South Wales, NICTA, and Open Kernel Labs</i>	
ShadowNet: A Platform for Rapid and Safe Network Evolution	29
<i>Xu Chen and Z. Morley Mao, University of Michigan; Jacobus Van der Merwe, AT&T Labs—Research</i>	

Networking

Design and Implementation of TCP Data Probes for Reliable and Metric-Rich Network Path Monitoring	43
<i>Xiapu Luo, Edmond W.W. Chan, and Rocky K.C. Chang, The Hong Kong Polytechnic University, Hong Kong</i>	
StrobeLight: Lightweight Availability Mapping and Anomaly Detection	57
<i>James W. Mickens, John R. Douceur, and William J. Bolosky, Microsoft Research; Brian D. Noble, University of Michigan</i>	
Hashing Round-down Prefixes for Rapid Packet Classification	71
<i>Fong Pong, Broadcom Corp.; Nian-Feng Tzeng, Center for Advanced Computer Studies, University of Louisiana at Lafayette</i>	

File and Storage Systems

Tolerating File-System Mistakes with EnvyFS	87
<i>Lakshmi N. Bairavasundaram, NetApp; Inc.; Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison</i>	
Decentralized Deduplication in SAN Cluster File Systems	101
<i>Austin T. Clements, MIT CSAIL; Irfan Ahmad, Murali Vilayannur, and Jinyuan Li, VMware, Inc.</i>	
FlexFS: A Flexible Flash File System for MLC NAND Flash Memory	115
<i>Sungjin Lee, Keonsoo Ha, Kangwon Zhang, and Jihong Kim, Seoul National University, Korea; Junghwan Kim, Samsung Electronics, Korea</i>	
Layering in Provenance Systems	129
<i>Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor, Harvard School of Engineering and Applied Sciences</i>	

Thursday, June 18

Distributed Systems

Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads.....	143
<i>Jeff Terrace and Michael J. Freedman, Princeton University</i>	
Census: Location-Aware Membership Management for Large-Scale Distributed Systems.....	159
<i>James Cowling, Dan R.K. Ports, Barbara Liskov, and Raluca Ada Popa, MIT CSAIL; Abhijeet Gaikwad, École Centrale Paris</i>	
Veracity: Practical Secure Network Coordinates via Vote-based Agreements.....	173
<i>Micah Sherr, Matt Blaze, and Boon Thau Loo, University of Pennsylvania</i>	

Kernel Development

Decaf: Moving Device Drivers to a Modern Language.....	187
<i>Matthew J. Renzelmann and Michael M. Swift, University of Wisconsin—Madison</i>	
Rump File Systems: Kernel Code Reborn	201
<i>Antti Kantee, Helsinki University of Technology</i>	
CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems	215
<i>Daniel Lohmann, Wanja Hofer, and Wolfgang Schröder-Preikschat, FAU Erlangen—Nuremberg; Jochen Streicher and Olaf Spinczyk, TU Dortmund</i>	

Automated Management

Automatically Generating Predicates and Solutions for Configuration Troubleshooting.....	229
<i>Ya-Yunn Su, NEC Laboratories America; Jason Flinn, University of Michigan</i>	
JustRunIt: Experiment-Based Management of Virtualized Data Centers.....	243
<i>Wei Zheng and Ricardo Bianchini, Rutgers University; G. John Janakiraman, Jose Renato Santos, and Yoshio Turner, HP Labs</i>	
vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities.....	259
<i>Byung Chul Tak, Pennsylvania State University; Chunqiang Tang and Chun Zhang, IBM T.J. Watson Research Center; Sriram Govindan and Bhuvan Urgaonkar, Pennsylvania State University; Rong N. Chang, IBM T.J. Watson Research Center</i>	

Short Papers

The Restoration of Early UNIX Artifacts	273
<i>Warren Toomey, Bond University</i>	
Block Management in Solid-State Devices	279
<i>Abhishek Rajimwale, University of Wisconsin, Madison; Vijayan Prabhakaran and John D. Davis, Microsoft Research, Silicon Valley</i>	
Linux Kernel Developer Responses to Static Analysis Bug Reports	285
<i>Philip J. Guo and Dawson Engler, Stanford University</i>	
Hardware Execution Throttling for Multi-core Resource Management	293
<i>Xiao Zhang, Sandhya Dwarkadas, and Kai Shen, University of Rochester</i>	

Friday, June 19

System Optimization

Reducing Seek Overhead with Application-Directed Prefetching	299
<i>Steve VanDeBogart, Christopher Frost, and Eddie Kohler, UCLA</i>	
Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances	313
<i>Anton Burtsev, University of Utah; Kiran Srinivasan, Prashanth Radhakrishnan, Lakshmi N. Bairavasundaram, Kaladhar Voruganti, and Garth R. Goodson, NetApp, Inc.</i>	
STOW: A Spatially and Temporally Optimized Write Caching Algorithm	327
<i>Binny S. Gill and Michael Ko, IBM Almaden Research Center; Biplob Debnath, University of Minnesota; Wendy Belluomini, IBM Almaden Research Center</i>	

Web, Internet, Data Center

Black-Box Performance Control for High-Volume Non-Interactive Systems	341
<i>Chunqiang Tang, IBM T.J. Watson Research Center; Sunjit Tara, IBM Software Group, Tivoli; Rong N. Chang and Chun Zhang, IBM T.J. Watson Research Center</i>	
Server Workload Analysis for Power Minimization using Consolidation	355
<i>Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari, IBM India Research Lab</i>	
RCB: A Simple and Practical Framework for Real-time Collaborative Browsing	369
<i>Chuan Yue, Zi Chu, and Haining Wang, The College of William and Mary</i>	

Bugs and Software Updates

The Beauty and the Beast: Vulnerabilities in Red Hat's Packages	383
<i>Stephan Neuhaus, Università degli Studi di Trento; Thomas Zimmermann, Microsoft Research</i>	
Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction	397
<i>Kristis Makris and Rida A. Bazzi, Arizona State University</i>	
Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation	411
<i>Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff, Purdue University</i>	

Message from the Program Co-Chairs

Welcome to the 2009 USENIX Annual Technical Conference!

Once again USENIX Annual Tech continues its tradition of high-quality papers that both break new ground and provide practical insight into modern computer systems. The program committee accepted 32 excellent papers—28 long and 4 short—selected from 191 submissions. The topics range from tolerating file system errors, tracking data provenance, and reprogramming sensor networks to breathing new life into UNIX artifacts. The program also features two engaging plenary talks. James Hamilton from Amazon opens the conference with a keynote on the critical issue of power in large-scale datacenters, and Hugo Award-winning author David Brin closes the conference with a plenary on new tools for problem solving in the information age and their policy implications in the near future.

We had an excellent program committee of 26 members drawn from both industry and academia. As with other conferences receiving a large number of submissions, the PC reviewed the submitted papers in two rounds. In the first round, every paper received at least three reviews. Based on these reviews, we assigned at least one additional review for 10 short papers and two additional reviews for 80 long papers. During the process, we also relied upon the specific expertise of 52 external reviewers. In total, the 191 submitted papers received 743 reviews. The program committee then met on March 12, 2009, for a lively full-day discussion that resulted in the final conference program. The PC shepherded all accepted papers, and the authors produced the polished final papers that constitute these proceedings.

As program co-chairs, we stand on the shoulders of many who did a tremendous amount of hard work. First, and most importantly, we thank all of the authors for their creative and compelling research—the conference and the community around which it forms would not exist without it. We thank the program committee for their diligence and commitment—each PC member reviewed nearly 30 papers, a dedicated undertaking. We thank our external reviewers for contributing much-needed reviews on short notice, Microsoft for hosting the PC meeting in Redmond, and Eddie Kohler for his tireless efforts supporting the HotCRP conference management software. Finally, we thank the USENIX staff—Ellie, Jane-Ellen, Devon, Casey, and many others—for all the tremendous behind-the-scenes work that makes the conference both a success and a reality.

Finally, we would like to thank our industry sponsors for their support in making the 2009 USENIX Annual Technical Conference possible and enjoyable. In particular we thank VMware, Google, Hewlett-Packard, Microsoft Research, NetApp, and Berkeley Communications for their generous support.

We hope you enjoy the program and the conference!

**Alec Wolman, Microsoft Research
Geoffrey M. Voelker, University of California, San Diego
2009 USENIX Annual Technical Conference Co-Chairs**

Satori: Enlightened page sharing

Grzegorz Miłoś, Derek G. Murray, Steven Hand
*University of Cambridge Computer Laboratory
Cambridge, United Kingdom*
First.Last@cl.cam.ac.uk

Michael A. Fetterman
*NVIDIA Corporation
Bedford, Massachusetts, USA*
mafetter@nvidia.com

Abstract

We introduce *Satori*, an efficient and effective system for sharing memory in virtualised systems. *Satori* uses *enlightenments* in guest operating systems to detect sharing opportunities and manage the surplus memory that results from sharing. Our approach has three key benefits over existing systems: it is better able to detect short-lived sharing opportunities, it is efficient and incurs negligible overhead, and it maintains performance isolation between virtual machines.

We present *Satori* in terms of hypervisor-agnostic design decisions, and also discuss our implementation for the Xen virtual machine monitor. In our evaluation, we show that *Satori* quickly exploits up to 94% of the maximum possible sharing with insignificant performance overhead. Furthermore, we demonstrate workloads where the additional memory improves macrobenchmark performance by a factor of two.

1 Introduction

An operating system can almost always put more memory to good use. By adding more memory, an OS can accommodate the working set of more processes in physical memory, and can also cache the contents of recently-loaded files. In both cases, cutting down on physical I/O improves overall performance. We have implemented *Satori*, a novel system that exploits opportunities for saving memory when running on a virtual machine monitor (VMM). In this paper, we explain the policy and architectural decisions that make *Satori* efficient and effective, and evaluate its performance.

Previous work has shown that it is possible to save memory in virtualised systems by sharing pages that have identical [23] and/or similar [4] contents. These systems were designed for unmodified operating systems, which impose restrictions on the sharing that can be achieved. First, they detect sharing opportunities by periodically scanning the memory of all guest VMs. The scanning rate is a trade-off: scanning at a higher rate detects more sharing opportunities, but uses more of the CPU. Secondly, since it overcommits the physical memory available to guests, the VMM must be able to page guest memory to and from disk, which can lead to poor performance.

We introduce *enlightened page sharing* as a collection of techniques for making informed decisions when sharing memory and distributing the benefits. Several projects have shown that the performance of a guest OS running on a VMM improves when the guest is modified to exploit the virtualised environment [1, 25]. In *Satori*, we add two main *enlightenments* to guests. We modify the virtual disk subsystem, to implement *sharing-aware block devices*: these detect sharing opportunities in the page cache immediately as data is read into memory. We also add a *repayment FIFO*, through which the guest provides pages that the VMM can use when sharing is broken. Through our modifications, we detect the majority of sharing opportunities much sooner than a memory scanner would, we obviate the run-time overhead of scanning, and we avoid paging in the VMM.

We also introduce a novel approach for distributing the benefits of page sharing. Each guest VM receives a *sharing entitlement* that is proportional to the amount of memory that it shares with other VMs. Therefore, the guests which share most memory receive the greatest benefit, and so guests have an incentive to share. Moreover, this maintains strong isolation between VMs: when a page is unshared, only the VMs originally involved in sharing the page are affected.

When we developed *Satori*, we had two main goals:

Detect short-lived sharing: We show in the evaluation that the majority of sharing opportunities are short-lived and do not persist long enough for a memory scanner to detect them. *Satori* detects sharing opportunities immediately when pages are loaded, and quickly passes on the benefits to the guest VMs.

Detect sharing cheaply: We also show that *Satori*'s impact on the performance of a macrobenchmark—even without the benefits of sharing—is insignificant. Furthermore, when sharing is exploited, we achieve improved performance for some macrobenchmarks, because the guests can use the additional memory to cache more data.

The rest of this paper is organised as follows. We begin by discussing the issues of memory management in both operating systems and virtualised platforms (Section 2). We then survey related systems (Section 3).

We present Satori in two parts: first, we justify the major design decisions that differentiate Satori from other systems (Section 4), then we describe how we implemented a prototype of Satori for the Xen VMM (Section 5). Finally, we perform a thorough evaluation of Satori’s performance, including its effectiveness at finding sharing opportunities and its impact on overall performance (Section 6).

2 Background

The problem of memory management has a long history in operating systems and virtual machine monitors. In this section, we review common techniques for managing memory as a shared resource (§ 2.1). We then describe the relevant issues for page sharing in virtual machine monitors (§ 2.2). Finally, we describe how paravirtualisation is used to improve performance in virtualised systems (§ 2.3).

2.1 Virtual memory management

Physical memory is a scarce resource in an operating system. If more memory is available, it can be put to good use, for example by obviating the need to swap pages to disk, or by caching recently-accessed data from secondary storage. Since memory access is several orders of magnitude faster than disk access, storing as much data as possible in memory has a dramatic effect on system performance.

Memory resource management was first formalised for operating systems by Denning in 1968, with the introduction of the *working set model* [3]. The working set of a process at time t is the set of pages that it has referenced in the interval $(t - \tau, t)$. This is a good predictor of what pages should be maintained in memory. Pages can then be allocated to each process so that its working set can fit in memory.

Since it is challenging to calculate the working set and τ parameter exactly, an alternative approach is to monitor the *page fault frequency* for each process [16]. If a process causes too many page faults, its allocation of pages is increased; and vice versa. This ensures acceptable progress for all processes.

OS-level approaches are inappropriate for a virtualised system. One of the key benefits of virtualisation is that it provides resource isolation between VMs. If the size of a VM’s working set or its page fault rate is allowed to determine its memory allocation, a malicious VM can receive more than its fair share by artificially inflating either measure. Instead, in our approach, we give a static allocation of physical memory to each VM, which provides strong performance isolation [1]. As we describe in § 4.2, our system provides surplus memory to VMs that participate in sharing. Our approach follows previous work on *self-paging*, which required each

application to use its own resources (disk, memory and CPU) to deal with its own memory faults [5].

2.2 Memory virtualisation and sharing

A conventional operating system expects to own and manage a range of contiguously-addressed physical memory. Page tables in these systems translate *virtual addresses* into *physical addresses*. Since virtualisation can multiplex several guest operating systems on a single host, not all guests will receive such a range of physical memory. Furthermore, to ensure isolation, the VMM’s and guests’ memory must be protected from each other, so the VMM must ensure that all updates to the hardware page tables are valid.

Therefore, a virtualised system typically has three classes of address. *Virtual addresses* are the same as in a conventional OS. Each VM has a *pseudo-physical address space*, which is contiguous and starts at address zero. Finally, *machine addresses* refer to the physical location of memory in hardware. A common arrangement is for guests to maintain page tables that translate from virtual to pseudo-physical addresses, and the VMM to maintain separate *shadow page tables* that translate directly from virtual addresses to machine addresses [23]. A more recent approach is to use additional hardware to perform the translation from pseudo-physical addresses to machine addresses [10, 19]. Finally, it is also possible to modify the OS to use machine addresses and communicate with the VMM to update the hardware page tables explicitly [1].

Pseudo-physical addresses provide an additional layer of indirection that makes it possible to share memory between virtual machines. Since, for each VM, there is a pseudo-physical-to-machine (P2M) mapping, it is possible to make several *pseudo-physical frame numbers* (PFNs) map onto a single *machine frame number* (MFN). Therefore, if two VMs each have a page with the same contents, the VMM can update the P2M mapping and the shadow page tables to make those pages use the same machine frame. We discuss how other systems detect duplicates in Section 3, and the Satori approach in Section 4.

If two VMs share a page, an attempt to write to it must cause a page fault. This is achieved by marking the page read-only in the shadow page table. Such a page fault is called a *copy-on-write fault*. When this occurs, the VMM handles the fault by allocating a new frame and making a private copy of the page for the faulting guest. It also updates the P2M mapping and shadow page tables to ensure that the guest now uses the private copy.

A consequence of page sharing is that the memory used by a VM can both dynamically decrease (when a sharing opportunity is exploited) and dynamically increase (when sharing is broken). This presents

a resource allocation problem for the VMM. A conventional operating system does not have fine-grained, high-frequency mechanisms to deal with memory being added or removed at run time (*Memory hotplug* interfaces are unsuitable for frequent, page-granularity addition and removal [13]). Therefore, one option is to use a *balloon driver* in each guest, which pins physical memory within a guest and donates it back to the VMM [1, 23]. The “balloon” can inflate and deflate, which respectively decreases and increases the amount of physical memory available to a given VM.

However, a balloon driver requires cooperation from the guest: an alternative is *host paging*, whereby the VMM performs page replacement on guests’ pseudo-physical memory [4, 23]. Host paging is expensive, because a VM must be paused while evicted pages are faulted in, and even if the VMM-level and OS-level page replacement policies are perfectly aligned, double paging (where an unused page must be paged in by the VMM when the OS decides to page it out) negatively affects performance. We deliberately avoid using host paging, and use a combination of the balloon driver (see § 4.2) and volatile pages (see § 4.3) to vary the memory in each guest dynamically.

Collaborative memory management (CMM) attempts to address the issue of double paging [14]. This system was implemented for Linux running on IBM’s z/VM hypervisor for the zSeries architecture. In CMM, the guest VM provides hints to the VMM that suggest what pages are being used, and what pages may be evicted with little penalty. In Satori, we use part of this work for a different purpose: instead of using hints to improve a host pager, we use them to specify pages which may be reclaimed when sharing is broken (see § 5.3).

2.3 Enlightenment

Our approach to memory sharing is based on *enlightenments*, which involve making modifications to operating systems in order to achieve the best performance in a virtualised environment; in this paper we use the terms “enlightenment” and “paravirtualisation” interchangeably. Operating systems have been modified to run on VMMs for almost as long as VMMs have existed: the seminal VM/370 operating system employs *handshaking* to allow guests to communicate with the VMM for efficiency reasons [15]. “Paravirtualisation” was coined for the Denali VMM [25], and Xen was the first VMM to run paravirtualised commodity operating systems, such as Linux, BSD and Windows [1]. Xen showed that by paravirtualising the network and block devices, rather than relying on emulated physical hardware, it was possible to achieve near-native I/O speeds.

More extreme paravirtualisation has also been proposed. For example, Pfaff *et al.* designed Ventana

as a *virtualisation-aware file system*, to replace virtual block devices as the storage primitive for one or more VMs [12]. This design concentrates on adding functionality to the file system—for example versioning, isolation and encapsulation—and considers sharing from the point of view of files shared between users. It does not specifically address resource management or aim to improve performance. Our approach is orthogonal to Ventana, and similar memory sharing benefits could be achieved with a virtualisation-aware file system. Indeed, using a system like Ventana would probably make it easier to identify candidates for sharing, and improve the overall efficiency of our approach.

Other systems, such as VMware ESX Server [23] and the Difference Engine [4] have a design goal of supporting unmodified guest OSs. In contrast, we have concentrated on paravirtualised guests for two reasons. First, there is an increasing trend towards enlightenments in both Linux and Microsoft Windows operating systems [18, 20]. Secondly, we believe that where there is a compelling performance benefit in using enlightenments, the necessary modifications will filter down into the vanilla releases of these OSs.

3 Related Work

Waldspurger described a broad range of memory management techniques employed in the VMware ESX Server hypervisor, including page sharing [23]. In VMware ESX Server, page sharing opportunities are discovered by periodically scanning the physical memory of each guest VM, and recording fingerprints of each page. When the scanner observes a repeated fingerprint, it compares the contents of the relevant two pages, and shares them if they are identical. In the same work, Waldspurger introduced the balloon driver that is used to alter guest memory allocations. However, since VMware ESX Server is designed to run unmodified guest operating systems, it must also support host paging. In Satori, we avoid host paging because of its negative performance impact (see § 2.2), and avoid memory scanning because it does not detect short-lived sharing opportunities (see § 4.1).

A contemporary research project has added page sharing to the Xen Virtual Machine Monitor. Vrable *et al.* began this effort with Potemkin [22], which uses *flash cloning* and *delta virtualization* to enable a large number of mostly-identical VMs on the same host. Flash cloning creates a new VM by copying an existing reference VM image, while delta virtualization provides copy-on-write sharing of memory between the original image and the new VM. Kloster *et al.* later extended this work with a memory scanner, similar to that found in VMware ESX Server [7]. Finally, Gupta *et al.* implemented the Difference Engine, which uses patching and compression to

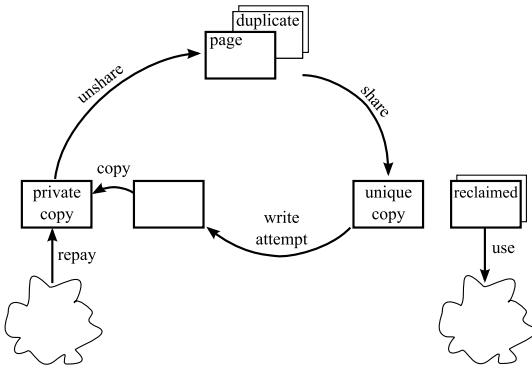


Figure 1: Sharing cycle

achieve greater memory savings than sharing alone. We have implemented Satori on Xen in a parallel effort, but we use guest OS enlightenments to reduce the cost of duplicate detection and memory balancing.

The Disco VMM includes some work on *transparent page sharing* [2]. In Disco, reading from a special *copy-on-write disk* involves checking to see if the same block is already present in main memory and, if so, creating a shared mapping to the existing page. We apply a similar policy for duplicate detection, as described in § 4.1. However, we also implement content-based sharing for disk I/O (§ 5.2), so it is not necessary to use copy-on-write disks, and furthermore we can exploit identical blocks within the same disk.

4 Design decisions

In this section, we present the major design decisions that differentiate Satori from previous work on page sharing [23, 4]. Figure 1 shows the life-cycle of a page that participates in sharing. This diagram raises three key questions, which we address in this section:

How are duplicates detected? We use *sharing-aware block devices* as a low-overhead mechanism for detecting duplicate pages. Since a large amount of sharing originates within the page cache, we monitor data as it enters the cache (§ 4.1).

How are memory savings distributed? When n identical pages are discovered, these can be represented by a single physical page, and $n-1$ pages are saved. We distribute these savings to guest VMs in proportion with their contribution towards sharing (§ 4.2).

What if sharing is broken? Shared pages are necessarily read-only. When a guest VM attempts to write to a shared page, the hypervisor makes a writable private copy of the page for the guest. We require that the guest itself provides a list of *volatile* pages that may be used to provide the necessary memory for private copies. In addition, we obviate the need for copying in certain cases (§ 4.3).

We have taken care to ensure that our answers to the above questions are hypervisor-agnostic and may be implemented together or individually. Although our prototype uses the Xen VMM (see Section 5), these techniques should also be useful for developers of other hypervisors. In particular, our duplicate detection and savings distribution policies could be implemented without modifying core OS components in the guest VMs. However, by enlightening the guest OS, it is possible to achieve better performance, and we contend that our techniques are best implemented as a whole.

4.1 How are duplicates detected?

In order to exploit page sharing, it is necessary to detect duplicate pages. As described in § 2.2, the most common approach to this problem is to scan the memory of all guest VMs periodically, and build up a list of page fingerprints that can be used to compare page contents. In this subsection, we propose *sharing-aware block devices* as a more efficient alternative. We discuss the problems with the scanning-based approach, and explain why the block interface is an appropriate point at which to detect potential sharing.

As we show in § 6.1, many sharing opportunities are short-lived, and yet these provide a large amount of overall sharing when taken as a whole. In principle, the memory scanning algorithm is exhaustive and all duplicates will eventually be found. However, in practise the rate of scanning has to be capped: in the extreme case, each memory write would trigger fingerprint recomputation. For example in VMware ESX Server the default memory scan frequency is set to once an hour, with a maximum of six times per hour [21]. Therefore, the theoretical mean *duplicate discovery time* for the default setting is 40min, which means that short-lived sharing opportunities will be missed. (We note that there are at least three relevant configuration options: the scan period, scan throughput (in MB per second per GHz of CPU), and maximum scan rate (in pages per second). In our evaluation (§ 6.1), the “aggressive” settings for VMware use the maximum for all three of these parameters.)

When an operating system loads data from disk, it is stored in the page cache, and other researchers have noted that between 63.8% and 93.0% of shareable pages between VMs are part of the page cache [8]. For example, VMs based on the same operating system will load identical program binaries, configuration files and data files. (In these systems, the kernel text will also be identical, but this is loaded by Xen domain builder (boot-loader), and does not appear in the page cache. Though we do not implement it here, we could modify the Xen domain builder to provide sharing hints.)

The efficacy of sharing-aware block devices relies

on the observation that, for the purpose of detecting duplicates, a good image of the page cache contents can be built up by observing the content of disk reads. While this approach does not capture any subsequent in-memory writes, we do not expect the sharing potential of dirty pages to be high. Since a VMM uses virtual devices to represent block devices, we have a convenient place to intercept block reads. We describe our implementation of sharing-aware block devices in § 5.2.

The situation improves further if several guests share a common base image for their block device. When deploying a virtualised system, it is common to use a single *substrate* disk for several VMs, and store VM-private modifications in a copy-on-write overlay file. If a guest reads a block from the read-only substrate disk, the block number is sufficient to identify it uniquely, and there is no need to inspect its contents. This scheme has the additional advantage that some reads can be satisfied without accessing the underlying physical device.

Previous work on page sharing emphasises zero pages as a large source of page duplicates. Clearly, these pages would not be found by block-device interposition. However, we take a critical view of zero-page sharing. An abundance of zero pages is often indicative of low memory utilisation, especially in operating systems which implement a scrubber. We believe that free page sharing is usually counterproductive, because it gives a false sense of memory availability. Consider the example of a lightly loaded VM, in which 50% of pages are zero pages. If these pages are reclaimed and used to run another VM, the original VM will effectively run with 50% of its initial allocation. If this is insufficient to handle subsequent memory demands, the second VM will have to relinquish its resources. We believe that free memory balancing should be explicit: a guest with low memory utilisation should have its allocation decreased. Several systems that perform this resource management automatically have been proposed [26].

4.2 How are memory savings distributed?

The objective of any memory sharing mechanism is to reuse the reclaimed pages in order to pay for the cost of running the sharing machinery. A common approach is to add the extra memory to a global pool, which can be used to create additional VMs [4]. However, we believe that only the VMs that participate in sharing should reap the benefits of additional memory. This creates an incentive for VMs to share memory, and prevents malicious VMs from negatively affecting the performance of other VMs on the same host. Therefore, Satori distributes reclaimed memory in proportion to the amount of memory that each VM shares.

When Satori identifies n duplicates of the same page, it will reclaim $n - 1$ redundant copies. In the common

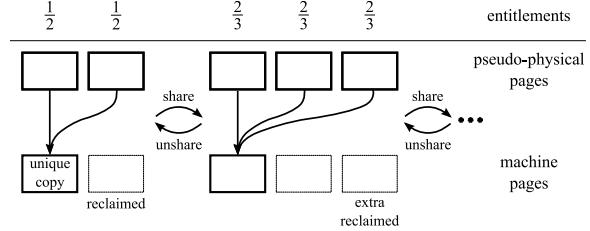


Figure 2: Sharing entitlement calculation

case of $n = 2$, our policy awards each of the contributing VMs with an entitlement of 0.5 pages—or, more generally, $\frac{n-1}{n}$ pages—for *each* shared page (Figure 2). For each page of physical memory, p , we define $n(p) \in \mathbb{N}$ as the *sharing rank* of that page. For VM i , which uses the set of pages $M(i)$, the total *sharing entitlement*, $s(i)$, is calculated as follows:

$$s(i) = \sum_{p \in M(i)} \frac{n(p) - 1}{n(p)}$$

Satori interrogates the sharing entitlements for each VM every second, and makes the appropriate amount of memory available to the VMs.

The sharing rank of a particular page will not necessarily remain constant through the lifetime of the sharing, since additional duplicates may be found, and existing duplicates may be removed. Therefore, the sharing entitlement arising from that page may change. Consider what happens when a new duplicate is discovered for an already n -way shared page. The VM that provided the new duplicate will receive an entitlement of $\frac{n}{n+1}$ pages, and the owners of the existing n duplicates will see their entitlement increase by $\frac{n}{n+1} - \frac{n-1}{n} = \frac{1}{n(n+1)}$ for each copy they own. Similarly, the entitlements must be adjusted when a shared page departs.

In Satori, guests claim their sharing entitlement using *memory balloons* [23]. When the entitlement increases, the balloon deflates and releases additional pages to the guest kernel. In our implementation we set up the guests to always claim memory as soon as it becomes available. However, guests can elect to use more complex policies. For example a guest may refrain from using its entitlement if it experiences low memory demand, or expects its shared pages to be short-lived. We have explicitly avoided using host paging to deal with fluctuating memory allocations. As a result, our implementation is simpler, and we have avoided the well-known problems associated with host paging. However, without host paging, we have to guarantee that the hypervisor can recover memory from the guests when it needs to create private copies of previously-shared pages. In the next subsection, we introduce the *repayment FIFO*, which addresses this issue.

4.3 What if sharing is broken?

If two or more VMs share the same copy of a page, and one VM attempts to write to it, the VMM makes a private copy of the page. Where does the VMM get memory for this copy?

Satori obtains this memory from a guest-maintained *repayment FIFO*, which contains a list of pages the guest is willing to give up *without* prior notification. The size of a VM’s repayment FIFO must be greater than or equal to its sharing entitlement. Our approach has three major advantages: (a) the hypervisor can obtain pages quickly, as there is no synchronous involvement with the guest, (b) there is no need for host paging, and (c) there is no risk that guest will be unable to relinquish resources due to double copy-on-write faults (i.e. a fault in the copy-on-write fault handler).

Pages in the repayment FIFO must not contain any irreplaceable information, because the guest will not have a chance to save their contents before the hypervisor reclaims them. Memory management subsystems already maintain book-keeping information about each page, which makes it possible to nominate such *volatile pages* without invasive changes.

In Satori the hypervisor uses sharing entitlements to determine the VM from which to reclaim memory. It does so by inspecting how much memory each VM drew from the sharing mechanism, in comparison to its current sharing entitlement. Since the sum of sharing entitlements is guaranteed to be smaller or equal to the number of removed duplicate pages, there will always be at least one VM with a negative memory balance (i.e. the VM drew more than its entitlement). Note that only the VMs which are involved in the broken sharing will be affected. This is essential to maintain performance isolation, as a malicious VM will be unable to affect any VMs with which it does not share memory.

A special case of broken sharing is when a page is reallocated for another purpose. For example, a guest may decide to evict a shared page from the page cache, scrub its content and reallocate it. In a copy-on-write system, the scrubber would cause a page fault when it begins to scrub the page, and the VMM would wastefully copy the old contents to produce a private version of the page. We use a scheme called *no-copy-on-write*, which informs the VMM that a page is being reallocated, and instead allocates a zero page (from a pre-scrubbed pool) for the private version.

To the best of our knowledge, Satori is the first system to address a covert channel created by memory sharing. An attacker can infer the contents of a page in another guest, by inducing sharing with that page and measuring the amount of time it takes to complete a write. (If a page has been shared, the processing of a copy-on-write fault will measurably increase the write latency.) For

example, we might want to protect the identity of server processes running in a guest, because security vulnerabilities might later be found in them. We allow guests to protect sensitive data by specifying which pages should never be shared. Any attempts to share with these pages will be ignored by the hypervisor.

5 Implementation

We implemented Satori for Xen version 3.1 and Linux version 2.6.18 in 11551 lines of code (5351 in the Xen hypervisor, 3894 in the Xen tools and 2306 in Linux). We chose Xen because it has extensive support for paravirtualised guests [1]. In this section, we describe how we implemented the design decisions from Section 4.

Our changes can be broken down into three main categories. We first modified the Xen hypervisor, in order to add support for sharing pages between VMs (§ 5.1). Next, we added support for sharing-aware block devices to the Xen control tools (§ 5.2). Finally, we enlightened the guest operating system, so that it can take advantage of additional memory and repay that memory when necessary (§ 5.3).

5.1 Hypervisor modifications

The majority of our changes were contained in the hypervisor. First of all, the upstream version of Xen does not support transparent page sharing between VMs, so it was necessary to modify the memory management subsystem. Once this support was in place, we added a hypercall interface that the control tools use to inform the hypervisor that pages may potentially be shared. Finally, we modified the page fault handler to deal with instances of broken sharing.

In § 2.2, we explained that each VM has a contiguous, zero-based pseudo-physical address space, and a P2M mapping for converting pseudo-physical addresses to machine addresses. To support transparent page sharing, it is necessary to allow multiple pseudo-physical pages to map to a single frame of machine memory. Furthermore, the machine frame that backs a given pseudo-physical page may change due to sharing. Therefore, it is simplest to use shadow page tables in the guest VMs. However, regular paravirtualised guests in Xen do not use shadow page tables, so we ported this feature from the code which supports fully-virtualised guests. In addition, we had to modify the reference counting mechanism used in Xen to keep track of page owners. In Xen each page has a single owner, so we added a synthetic “sharing domain” which owns all shared pages.

As described in § 5.3, we maintain information about the state of each (pseudo-)physical page in each guest. Both the guest and the hypervisor may update this information, so it is held in a structure that is shared between the hypervisor and the guest. The hypervisor uses this

structure to select which page should be used to satisfy a copy-on-write fault (either a page from the repayment FIFO, or, in the no-copy-on-write case, a zero-page).

We export the sharing functionality to the guest through the hypercall interface. We add three new hypercalls, named `share_mfns`, `mark_ro` and `get_ro_ref`.

The `share_mfns` hypercall takes two machine frame numbers (MFNs)—a source MFN and a client MFN—and informs the hypervisor that all pseudo-physical pages backed by the client frame should now use the source frame. The hypercall works as follows:

1. Mark the source and client frame as read-only, if they are not already.
2. Compare the contents of the source and client frame. If they are not equal, return an error.
3. Remove all mappings to the client MFN from the shadow page tables.
4. Update the relevant P2M mappings to indicate that the source frame should be used in place of the client frame.
5. Free the client frame for use by the guest VMs.

Note that this hypercall is not guaranteed to succeed. For example, after the duplicate detector notices that two pages are the same, but before they are marked read only, a guest might change the contents of one of the pages. Therefore, the hypercall may fail, but there is no risk that the contents of memory will be incorrect: the source and client frame will continue to be used as before.

For copy-on-write disks, we want to make an early decision about whether or not physical I/O will be required. Therefore, we use the `mark_ro` hypercall to enforce read-only status on all pages that are read from the read-only substrate. (Technically, we make a page read-only by treating it as 1-way shared; if the guest writes to it, the sharing is simply broken by marking the page as writable and changing the owner to the guest.) The complementary `get_ro_ref` hypercall ensures that the contents of the frame have not been changed (i.e. that the MFN is still read-only), and increments the sharing reference count to prevent it from being discarded. We describe the copy-on-write disk support in § 5.2.

The final hypervisor component required for page sharing is a modified page fault handler. We added two new types of page fault, which Xen must handle differently. The first is a straightforward *copy-on-write fault*, which is triggered when a guest attempts to write to a shared page. In this case, the handler recalculates the sharing entitlements for the affected guests, and reclaims a page from one of the guests that now has claimed more memory than its entitlement. The handler removes this page from the appropriate guest’s repayment FIFO and copies in the contents of the faulting page. We also add

a *discard fault*, which arises when a guest attempts to access a previously-volatile page that the VMM has reclaimed. If so, the handler injects this fault into the guest, as described in § 5.3.

5.2 Sharing-aware block devices

We implemented duplicate detection using *sharing-aware block devices*. Xen provides a high-performance, flexible interface for block I/O using *split devices*. The guest contains a *front-end driver*, which presents itself to the guest OS as a regular block device, while the control VM hosts a corresponding *back-end driver*. Previous work has shown how the back-end is a suitable interposition point for various applications [24], in particular for creating a distributed storage system [9]. We use the existing *block-tap* architecture to add duplicate detection.

The key steps in a block-tap read request are as follows:

1. The front-end (in the guest) issues a read request to the back-end through the inter-VM *device channel*, by providing a block number and an I/O buffer.
2. The back-end maps the I/O buffer into a user-space control tool, called `tapdisk`.
3. `tapdisk` performs device-specific processing for the given block number, and returns control to the back-end driver.
4. The back-end unmaps the I/O buffer and notifies the front-end of completion.

Since `tapdisk` is implemented as a user-space process and provides access to I/O data, it is simple to add custom block-handling code at this point. Satori modifies the `tapdisk` read path in order to record information about what data is loaded into which locations in the guests’ memory. We developed two versions of duplicate detection: content-based sharing, and copy-on-write disk sharing.

For content-based sharing, we hash the contents of each block as it is read from disk. We use the hash as the key in a hashtable, which stores mappings from hash values to machine frame numbers (MFNs). First, we look for a match in this table, and, if this is successful, the resulting MFN is a candidate for sharing with the I/O buffer. Note that the MFN is merely a hint: the contents of that frame could have changed, but since we have already loaded the data into the I/O buffer, it is acceptable for the sharing attempt to fail. If the hash is not present in the hashtable, we invalidate any previous entry that maps to the I/O buffer’s MFN, and store the new hash-to-MFN mapping.

For copy-on-write disk sharing, the process is slightly different (see Figure 3). The first time a block is read from the substrate disk, Satori invokes the `mark_ro` hypercall on that page, and stores a mapping from the

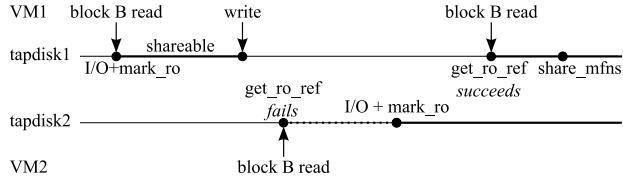


Figure 3: `mark_ro` and `get_ro_ref` usage for copy-on-write disks.

block number to the I/O buffer MFN. (If the guest subsequently writes to the page before it is shared, the read-only status is removed.) On subsequent reads, Satori consults the block number-to-MFN mapping to see if the block is already cached in memory. If it is, Satori invokes the `get_ro_ref` hypercall on the MFN, which, if it succeeds, ensures that the subsequent call to `share_mfn`s will be successful. If the block number is not found in the mapping table or if `get_ro_ref` fails, Satori must request physical disk I/O to read the appropriate block. At this point, a second look-up could be used to detect content-based sharing opportunities.

The Xen architecture places each virtual block device in a separate `tapdisk` process, to simplify management and improve fault isolation. However, we need to share information between devices, so it was necessary to add a single additional process, called `spctrl` (**S**hared **P**age **C**ache **CO**nT**R**oller), which hosts the mappings between content hashes or block numbers, and MFNs. The `tapdisk` processes communicate with `spctrl` using pipes, and our current implementation of `spctrl` is single-threaded.

5.3 Guest enlightenments

In Satori, we have used enlightenments to obtain OS-level information about guest pages. These extend the existing paravirtualised Linux guests, which Xen already supports [1].

For Satori, the most important enlightenment is adding the repayment FIFO to the guest kernel. Recall that the repayment FIFO is a list of *volatile pages*, i.e. physical pages that the operating system is willing to relinquish at any time (in particular, when sharing is broken and a new frame is needed for a private copy). Since the guest must relinquish these pages without warning, it is essential that their contents can be reconstructed from another source. Hence an obvious source of volatile pages is the set of clean pages in the page cache. We paravirtualised the Linux page cache to provide *page hints* about volatile pages.

We based our implementation of volatile pages on earlier work on Collaborative Memory Management (CMM) [14]. CMM is, in essence, a memory controller which relies on page states (especially page volatility) to dynamically adjust the available (machine) memory

for each guest. CMM is implemented for the IBM zSeries z/VM hypervisor, but the majority of the code is architecture-independent, as it deals with page-state transitions in the Linux page and swap caches. We built on CMM’s page hinting by adding support for the x86 architecture and the Xen hypervisor.

The major difference between x86 and zSeries (s390) in the context of volatile pages is the handling of dirtying. The x86 architecture maintains dirty bits for virtual pages in the PTE, whereas the s390 architecture maintains a dirty bit for each machine page. Since a given page can only become volatile if it is not dirty, we implemented a machine-page-level dirty bit in software for the x86 architecture. Our approach is more conservative than is strictly necessary, because we consider the existence of any writable mapping to dirty the page, even if there was no actual write.

Satori uses a shared structure between Xen and each guest to store and modify page states (as discussed in § 5.1). The page states read from this structure are used in the guest page fault handler to distinguish between “regular” and *discard* faults. On a discard fault, Linux uses reverse mappings to remove all references to the discarded page, effectively removing the page from its respective cache (page or swap).

We also use the instrumentation in the page allocator, already present in order to drive page state transitions, to support the no-copy-on-write policy. Whenever a page is reallocated, we update the shared page state structure to reflect this. On a write fault to a shared page, Xen checks to see whether the page has been reallocated, and, if so, provides a page from its zero page cache.

In addition, we have added support to guests for specifying that some pages must not be shared (to avoid the secret-stealing attack described in § 4.3). At present, we allow the guest to specify that a set of pseudo-physical pages must never be shared (i.e. all calls to `share_mfn`s or `get_ro_ref` will fail).

6 Evaluation

To characterise Satori’s performance, we have conducted an evaluation in three parts. First, we have profiled the opportunities for page sharing under different workloads (§ 6.1). In contrast with previous work, we specifically consider the *duration* of each sharing opportunity, as this is crucial to the utility of page sharing. We then measure the effectiveness of Satori, and show that it is capable of quickly detecting a large amount of sharing (§ 6.2). Finally, we measure the effect that Satori has on performance, in terms of the benefit when sharing is enabled, and the overhead on I/O operations (§ 6.3).

For our tests we used two Dell PowerEdge 1425 servers each equipped with two 2.8 GHz Xeon CPUs, 2.5 GB of RAM and an 80 GB Seagate SATA disk. VMs

Rank	Pages saved	Percentage saving
2	1565421	79.7%
3	137712	7.01%
4	59790	3.04%
5	18760	0.96%
6	24850	1.27%
8	10059	0.51%
10	10467	0.53%
14	10218	0.52%
others	126865	6.46%

Table 1: Breakdown of sharing opportunities by rank (excluding zero-pages).

ran Ubuntu Linux 8.04 in all cases, except for two experiments, for which we state the OS version explicitly.

In the following subsections, we make repeated reference to several workloads, which we abbreviate as follows:

KBUILD-256 Vanilla Linux 2.6.24 kernel build with 256 MB of physical memory.

KBUILD-512 As KBUILD-256, with 512 MB.

HTTPERF httpperf benchmark [6] run against Apache web-server with 512 MB of memory, serving randomly generated static webpages.

RUBiS RUBiS web auction application with 512 MB, serving requests generated by the default client workload generator [11].

6.1 Sharing opportunities

The major difference between Satori and contemporary page sharing schemes is that it can share many identical pages as soon as they are populated. In this subsection, we show that a substantial proportion of sharing is short-lived. Therefore, Satori is much more likely to exploit this sharing than schemes that rely on periodically *scanning* physical memory, looking for identical page contents [23, 4].

To analyse the sharing opportunities, we ran each of the KBUILD-256, KBUILD-512, HTTPERF and RUBiS workloads in two virtual machines for 30 minutes, and took a memory dump every 30 seconds.

Table 1 shows the number of pages that can be freed using page sharing, for each rank. (In a sharing of rank n , n identical pages map to a single physical page.) The figures are an aggregate, based on the total of 60 memory dumps sampled from pairs of VMs running the KBUILD-512, HTTPERF and RUBiS workloads. Note that most sharing opportunities have rank 2: i.e. two identical pages exist and can be combined into a single physical page.

Operation	Count	Total (ms)	Avg (μ s)
mark_ro	127479	5634	44.1
share_mfns	61905	474	7.7
get_ro_ref	69124	64	0.9
Total	258508	6172	—

Table 2: Breakdown of Satori hypercalls during HTTPERF workload

Figure 4 compares the number of *unique* shared pages during the KBUILD-256 and KBUILD-512 workloads. (By considering only unique shared pages, we underestimate the amount of savings for pages with rank > 2. Table 1 demonstrates that the majority of shareable pages have rank 2, except zero pages, which we address separately below.) We have divided the sharing opportunities into four duration ranges. The figures clearly show that a substantial amount of sharing is short-lived, especially in a more memory-constrained setup (KBUILD-256). Also, the amount of sharing for the KBUILD-512 workload is approximately twice as much as that for KBUILD-256, because of less contention in the page cache. Finally, the kernel build process completes 6 minutes sooner with 512 MB of memory: this makes the benefits of additional memory clear.

Figure 5 separately categorises shareable non-zero pages and zero pages into the same duration ranges as Figure 4. It should be noted that the number of sharing opportunities arising from zero pages (Figures 5(c) and 5(d)) is approximately 20 times greater than from non-zero pages (Figures 5(a) and 5(b)). However, more than 90% of zero-page sharing opportunities exist for less than five minutes. This supports our argument that the benefits of zero-page sharing are illusory.

In § 4.1, we stated that, on average, it will take 40 minutes for VMware ESX Server to detect a duplicate page using its default page scanning options. We ran the following experiment to validate this claim. Two VMs ran a process which read the same 256 MB, randomly-generated file into memory, and Figure 6 shows the number of shared pages as time progresses. The lower curve, representing the default settings, shows that half of the file contents are shared after 37 minutes, which is close to our predicted value; the acceleration is likely due to undocumented optimisations in VMware ESX Server. The higher curve shows the results of the same experiment when using the most aggressive scanning options. Using the same analysis, we would expect a duplicate on average to be detected after 7 minutes. In our experiment, half the pages were detected after almost 20 minutes, and we suspect that this is a result of the aggressive settings causing the page hint cache to be flushed more often.

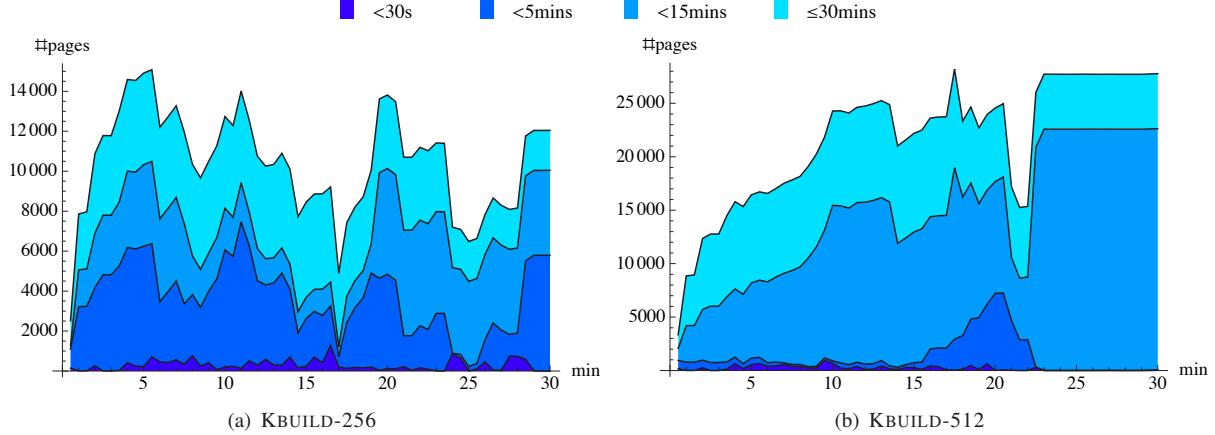


Figure 4: Sharing opportunities during the execution of workloads KBUILD-256 and KBUILD-512.

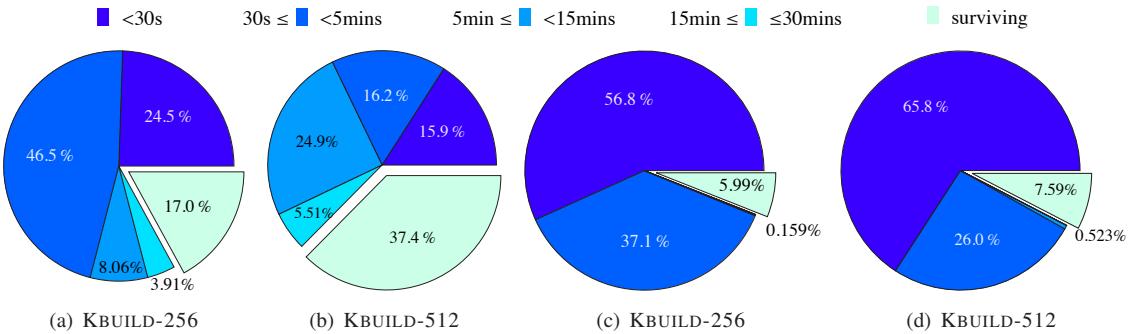


Figure 5: Duration of page sharing opportunities for kernel compilation workloads. (a) and (b) show non-zero pages, (c) and (d) zero pages. The exploded sectors show sharings left at the end of the experiment.

6.2 Satori effectiveness

In the next set of experiments, we measured the amount of sharing that Satori achieved using sharing-aware block devices. We also examined how the surplus memory was distributed between individual virtual machines.

The first experiment used two pairs of virtual machines. Two VMs each ran the HTTPPERF-256 workload, i.e. the HTTPPERF workload with 256 MB of memory (rather than 512 MB). Because the aggregate amount of memory was insufficient to cache the entire data set in memory, the number of shareable pages varied as data was loaded into and evicted from each VM’s page cache. The other two VMs each ran the KBUILD-512 workload; however they used Debian Linux rather than Ubuntu.

Figure 7 shows that the sharing entitlements for the VMs running KBUILD-512 are unaffected by the highly variable amount of sharing between the two HTTPPERF workloads. Also, because we used different OSes for each pair of VMs, the sharing entitlements achieved before the workloads started (5 to 6 minutes after the measurements began) differ by about 30%.

Next, we ran two instances of a workload in separate

VMs for 30 minutes, and repeated the experiment for the KBUILD-256, KBUILD-512, HTTPPERF and RUBIS workloads. We ran these experiments under Satori and measured the number of shared pages, and compared these to memory dumps using the same methodology as described in § 6.1.

Figure 8 summarises the amount of sharing that Satori achieves for each workload. Satori performs best with the HTTPPERF workload, shown in Figure 8(c). In this case, it achieves 94% of the total sharing opportunities, which is to be expected since HTTPPERF involves serving a large volume of static content over HTTP, and the majority of the data is read straight from disk. The RUBIS workload performs similarly, with Satori achieving 91% of the total. The kernel compilation workloads, KBUILD-256 and KBUILD-512, perform less well. KBUILD-512 achieves about 50% of the total sharing opportunities until the very end of the build, when the kernel image is assembled from individual object files. KBUILD-256 is more memory-constrained, which forces the OS to flush dirty (non-shareable) caches.

Finally, we ran two experiments which evaluated Satori in a more heterogeneous environment. In the

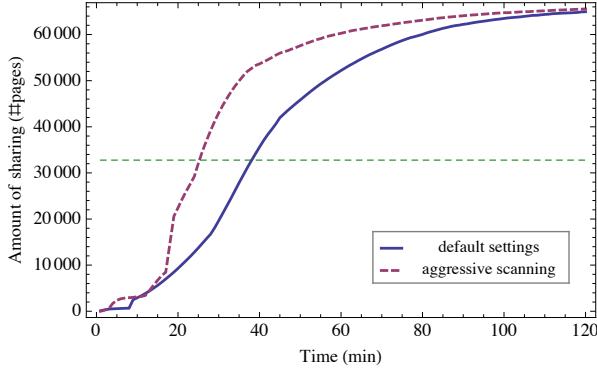


Figure 6: Sharing as time progresses for default and aggressive scanning settings in VMware ESX Server.

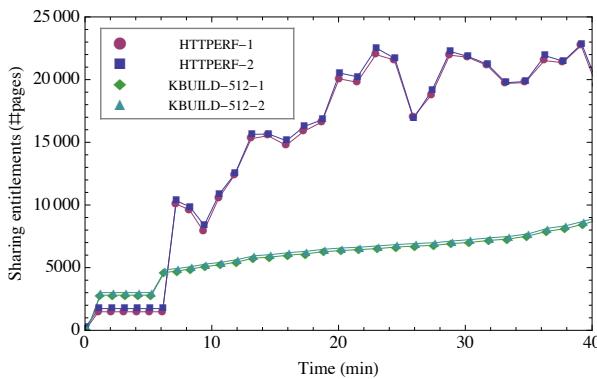


Figure 7: Sharing entitlements for two KBUILD-512 and two HTTPPERF-256 workloads executing simultaneously.

first experiment, two VMs running the same version of Ubuntu Linux performed the HTTPPERF and RUBIS workloads. In this setup Satori was able to exploit over 70% of the total sharing opportunities. (The remaining 30% was mostly due to the identical kernel images, which the current version of Satori does not detect.). In the second experiment, we used the same workloads with different guest OSs (Ubuntu and Debian respectively). In this setup, 11 MB of sharing was theoretically possible, and only because the two distributions use an identical kernel. In this case, Satori could only achieve approximately 1 MB of savings (9% of the total).

Although Satori achieves varying results in terms of memory savings, recall that these results come solely from using our enlightened block device. These results show that we can exploit up to 94% (for HTTPPERF) of the total sharing opportunities through this method alone. The alternative approach, which involves scanning memory, incurs a constant overhead at run-time, and must be rate-limited to prevent performance degradation [21]. The Difference Engine exhibits an overhead of up to 7% on some macrobenchmarks, though this includes the overhead of page compression and sub-page-

level patching [4]. Satori provides a flexible interface for adding other sharing policies: we are developing a tool that systematically identifies the source(s) of other sharing opportunities. We hope that this will lead to additional enlightenments that improve Satori’s coverage.

In § 4.3, we described an attack on memory sharing that allows a VM to identify sensitive data in another guest. On VMware ESX Server, we were able to determine the precise version of sshd running in another guest, by loading a page from each of 50 common distribution-supplied sshd binaries into memory, and periodically measuring the write latency to these pages. (On our hardware, we observed a 28-times increase for the matching page.) In Satori, we were able to protect the entire sshd address space, and, as a result, this attack failed.

6.3 Performance impact

We initially stated that memory sharing is desirable because it can improve the overall performance of VMs running on the same physical machine. In this subsection, we investigate the performance characteristics of Satori under various workloads. First, we measure *negative* impact: Satori introduces new operations for sharing memory, and these incur a measurable cost. We measure the cost of each sharing-related hypercall, and the overall effect on disk I/O. However, we then show that, for realistic macrobenchmarks, the overhead is insignificant, and the additional memory can improve overall performance.

To measure the cost of individual sharing-related operations, we instrumented the Xen hypervisor to record the number and duration of each hypercall. Table 2 shows the results for a 30-minute HTTPPERF workload. The first thing to note is that Satori-related operations account for less than 6.2 seconds of the 30-minute benchmark. Of the individual operations, `mark_ro` is the most expensive, as it must occasionally perform a brute-force search of the shadow page tables for all mappings of the page to be marked read-only. We could optimise performance in this case by making the guest VM exchange back-reference information with the hypervisor, but the overall improvement would be negligible.

Satori detects sharing by monitoring block-device reads, and therefore the majority of its overhead is felt when reading data from disk. In order to measure this overhead, and stress-test our implementation, we ran the Bonnie filesystem benchmark in a guest VM against a sharing-aware block device. Table 3 shows a breakdown of read bandwidths. We ran the benchmark in four configurations, and repeated each experiment five times. In the baseline configuration, we disabled all Satori mechanisms. In successive configurations, we enabled content hashing, IPC with `spctrl`, and finally hash lookup,

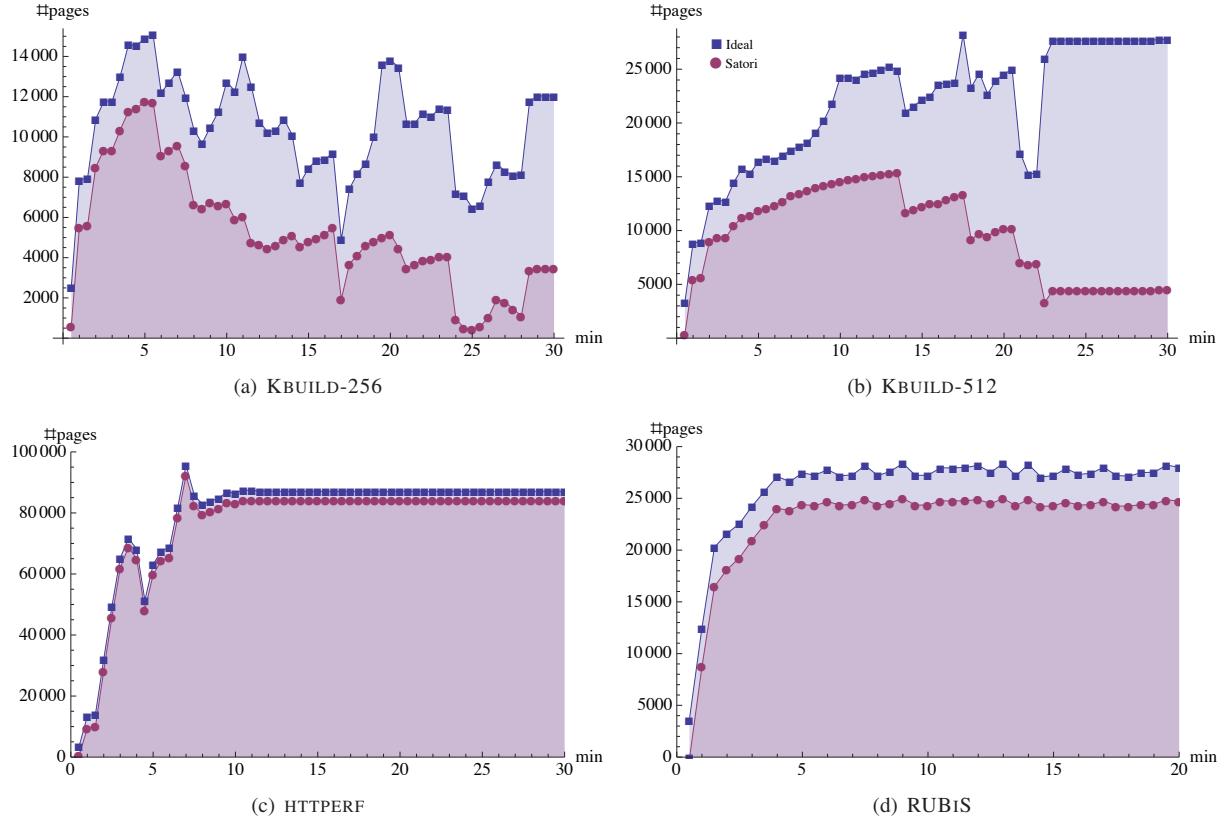


Figure 8: Amount of sharing achieved by Satori for each of the four main workloads (no zero-pages)

in order to isolate the performance impact of each function. Table 3 reports bandwidth figures for reads using `getc()`, and “intelligent reads”, which use a block size of 16384 bytes.

The first thing to note is that Bonnie performs *sequential* reads on a 512 MB file, so the effect of any computation on the I/O path is amplified. (The impact of Satori on random reads is negligible.) Therefore, the 34.8% overhead for chunked reads with Satori fully enabled is a worst-case figure, and is to be expected. With a realistic workload, the I/O cost is likely to dominate. Nevertheless, it is instructive to consider the individual overheads:

- The overhead involved in hashing is relatively constant and less than 0.4%.
- IPC with the `spctrl` process is costly. The present implementation uses UNIX pipes to communicate with `spctrl`, which involves two additional context switches per read request. We plan to redesign this component to store the hashtable in a shared memory segment.
- The relative overhead of fully-enabled Satori is worse in the chunked read case, because less time is being wasted in making repeated system calls in the guest VM.

While we are continuing to improve Satori’s performance, and eliminate these bottlenecks, we have only encountered the above issues when running Bonnie. For example, we ran a stripped-down kernel compilation in a single VM, which took an average of 780 seconds with Satori disabled, and 779 seconds with Satori fully enabled. Since the standard deviation over five runs was 27 seconds, it is clear that the overhead is statistically insignificant. In this experiment, the workload ran in isolation, and there were no benefits from sharing. As we will see next, the advantage of having additional memory can improve performance for many workloads.

We first ran an experiment to illustrate the benefit of memory sharing between VMs that share a copy-on-write disk. We ran a workload that read the contents of a copy-on-write disk into memory in a pseudorandom order. Five seconds later (while the first read was ongoing), we started the same workload, reading from the same disk in the same sequence, in another VM. Figure 9 shows the progress that both VMs achieved as a proportional gradient. VM1 reads at a consistent rate of 4.96 MB/s. When the workload commences in VM2, its initial rate is 111 MB/s, as the data that it reads can be provided by sharing memory with the page cache in VM1. After 0.22 seconds, VM2 has read all of the data held

Mode	Read bandwidth (MB/s)							
	getc()				“Intelligent read”			
	Min	Max	Avg	Overhead	Min	Max	Avg	Overhead
No sharing	26.9	28.2	27.6	—	47.1	47.4	47.4	—
Hashing only	26.1	28.4	27.5	0.4%	47.1	47.4	47.3	0.2%
Hashing + IPC	22.7	23.8	23.2	15.9%	31.8	33.0	32.4	31.6%
Sharing enabled	23.2	24.9	24.2	12.9%	30.7	31.1	30.9	34.8%

Table 3: Results of the Bonnie filesystem benchmark on Satori

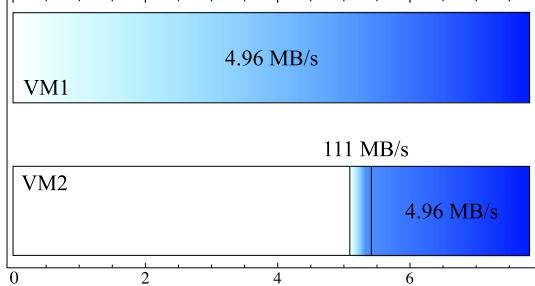


Figure 9: Copy-on-write disk read rates

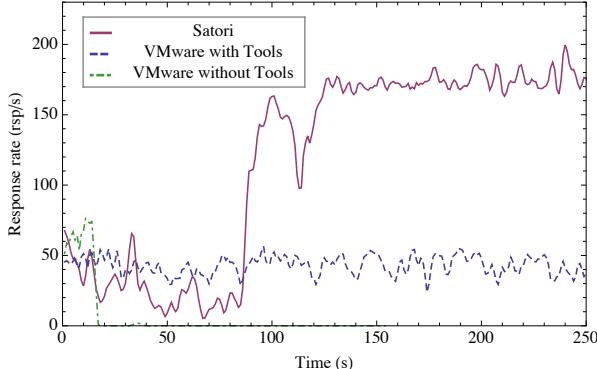


Figure 10: Aggregate HTTPPERF response rates for the two VMs running on Satori, VMware, and VMware with VMware Tools

by VM1, and the two VMs become mutually synchronised, at the original rate, which is limited by the disk access time. Although this example is artificial, it shows Satori’s effectiveness at exploiting page cache sharing for copy-on-write disks. Many recent cloud computing systems, such as Amazon’s EC2 [17], encourage the use of standard *machine image* files, which are natural candidates for a copy-on-write implementation. Satori would be particularly effective in this case.

Finally, we ran the HTTPPERF workload in two VMs as a macrobenchmark, to discover how well Satori exploits the extra memory that is made available through sharing. We compare Satori to VMware ESX Server—the leading commercial hypervisor—which uses the techniques described by Waldspurger to achieve page sharing and memory overcommitment [23].

Figure 10 shows how the aggregate HTTPPERF response rate changes over time for Satori and VMware (with and without VMware Tools). The performance of Satori can be divided into two phases. First, it achieves approximately 30 responses per second while the cache is being loaded, which takes approximately 85 seconds. The response rate then jumps to between 170 and 200 responses per second as all subsequent requests can be satisfied from caches. In order to maintain these response rates, the VMs use their sharing entitlements to increase their page cache sizes. The physical memory available to each VM grows to over 770 MB over the first 120 seconds of the experiment.

The results for VMware are interesting. We note first that it was necessary to install the VMware Tools (which include a balloon driver) in order to achieve performance that was comparable to Satori. Without the VMware Tools, the VMM begins paging after approximately 15 seconds, and throughput drops almost to zero. Once host paging starts, the throughput only recovers occasionally, and never to more than 5 responses per second. With the VMware Tools installed, we observed that balloon permanently limited each VMs physical memory allocation to 500 MB. Therefore, the VMs were able to make progress without host paging, but the data set did not fit in the cache, and the response rate remained at around 40 responses per second. VMware was unable to establish sufficient sharing because the lifetime of a page in either page cache was usually too short for the memory scanner to find it.

7 Conclusions

We described Satori, which employs enlightenments to improve the effectiveness and efficiency of page sharing in virtualised environments. We have identified several cases where the traditional page sharing approach (i.e. periodic memory scanning) does not discover or exploit opportunities for sharing. We have shown that, by using information from the guest VMs, and making small modifications to the operating systems, it is possible to discover a large fraction of the sharing opportunities with insignificant overhead.

Our implementation has concentrated on sharing-aware block devices. In the future we intend to add other

enlightened page sharing mechanisms—such as long-lived zero-page detection, page-table sharing and kernel text sharing—which will improve Satori’s sharing discovery rate. We also intend to investigate the application of our technique to nearly-identical pages [4].

Acknowledgments

We wish to thank members of the Systems Research Group at the University of Cambridge for the many fruitful discussions that inspired this work. We also wish to thank our shepherd, Geoffrey Voelker, and the anonymous reviewers for their insightful comments and suggestions that improved this paper.

References

- [1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.
- [2] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.
- [3] P. J. Denning. The working set model for program behavior. In *Proceedings of the 1st ACM Symposium on Operating System Principles*, 1967.
- [4] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *8th USENIX symposium on Operating System Design and Implementation*, 2008.
- [5] S. M. Hand. Self-paging in the nemesis operating system. In *Proceedings of the 3rd USENIX symposium on Operating Systems Design and Implementation*, 1999.
- [6] Hewlett-Packard Development Company, L.P. http://www.hpl.hp.com/research/linux/httpperf/, accessed 9th January, 2009.
- [7] J. F. Kloster, J. Kristensen, and A. Mejhlholm. On the Feasibility of Memory Sharing. Master’s thesis, Aalborg University, June 2006.
- [8] J. F. Kloster, J. Kristensen, and A. Mejhlholm. Determining the use of Interdomain Shareable Pages using Kernel Introspection. Technical report, Aalborg University, January 2007.
- [9] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Proceedings of the 3rd EuroSys conference on Computer Systems*, 2008.
- [10] G. Neiger, A. Santoni, F. Leung, D. Rogers, and R. Uhlig. Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel® Technology Journal*, 10(3):167–178, Aug 2006.
- [11] ObjectWeb Consortium. RUBiS – Home Page, 2008. <http://rubis.objectweb.org/>, accessed 9th January, 2009.
- [12] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd USENIX symposium on Networked Systems Design and Implementation*, 2006.
- [13] J. H. Schopp, K. Fraser, and M. J. Silbermann. Resizing Memory with Balloons and Hotplug. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.
- [14] M. Schwidfsky, H. Franke, R. Mansell, H. Raj, D. Osisek, and J. Choi. Collaborative Memory Management in Hosted Linux Environments. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.
- [15] L. H. Seawright and R. A. MacKinnon. VM/370 - A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.
- [16] A. S. Tanenbaum. *Modern Operating Systems*, page 122. Prentice-Hall, 1992.
- [17] (Unattributed). Amazon Elastic Compute Cloud (Amazon EC2). <http://aws.amazon.com/ec2/>, accessed 5th January, 2009.
- [18] (Unattributed). Understanding Full Virtualization, Paravirtualization and Hardware Assist. Technical report, VMWare, Inc., 2007.
- [19] (Unattributed). AMD-V™ Nested Paging. Technical report, Advanced Micro Devices, Inc., Jul 2008.
- [20] (Unattributed). Performance and capacity requirements for Hyper-V, 2008. <http://technet.microsoft.com/en-us/library/dd277865.aspx>, accessed 9th January 2009.
- [21] (Unattributed). *Resource Management Guide, ESX Server 3.5, ESX Server 3i version 3.5, VirtualCenter 2.5*, page 171. VMWare, Inc., 2008. http://www.vmware.com/pdf/vi3_35/esx_3_r35u2/vi3_35_25_u2_resource_mgmt.pdf, accessed 9th January, 2009.
- [22] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating systems Principles*, 2005.
- [23] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX symposium on Operating Systems Design and Implementation*, 2002.
- [24] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.
- [25] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th USENIX symposium on Operating Systems Design and Implementation*, 2002.
- [26] W. Zhao and Z. Wang. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, 2009.

vNUMA: A Virtual Shared-Memory Multiprocessor

Matthew Chapman^{*†}

Gernot Heiser^{*‡}

^{*}*The University of New South Wales*

[†]*NICTA*

[‡]*Open Kernel Labs*

matthewc@cse.unsw.edu.au

<http://ertos.nicta.com.au>

Abstract

vNUMA, for *virtual NUMA*, is a virtual machine that presents a cluster as a virtual shared-memory multiprocessor. It is designed to make the computational power of clusters available to legacy applications and operating systems.

A characteristic aspect of vNUMA is that it incorporates distributed shared memory (DSM) inside the hypervisor, in contrast to the more traditional approach of providing it in middleware. We present the design of vNUMA, as well as an implementation on Itanium-based workstations. We discuss in detail the enhancements to standard protocols that were required or enabled when implementing DSM inside a hypervisor, and discuss some of the tradeoffs we encountered. We examine the scalability of vNUMA on a small cluster, and analyse some of the design choices.

1 Introduction

Shared-memory multiprocessor (SMM) systems provide a simple programming model compatible with a large base of existing applications and operating systems. They naturally lend themselves to providing a single system image (SSI) running a single operating-system (OS) instance with a single resource name space.

However, for many compute-intensive applications, a network of commodity workstations presents a more cost-effective platform. These systems deliver the same (theoretical) compute power with much less expensive hardware, and are easily extensible and re-configurable. Yet their computing power is much more difficult to harness. Most existing OSes were not designed for cluster environments, and applications designed for shared-memory systems need to be redesigned for clusters by using explicit communication over the network.

Previous attempts have been made to bridge the gap between the ease of programming and legacy support of SMM systems and the economies of cluster hardware. These include distributed shared memory (DSM) libraries such as Ivy [23] or Treadmarks [19], which

provide a limited illusion of shared memory to applications, provided that the programmer uses the primitives supplied by the library. Other projects have attempted to retrofit support for cluster-wide process scheduling and migration into OSes [2, 27, 35]. However, these approaches require extensive and intrusive OS changes, which are difficult to keep up to date with the fast pace of OS development.

This paper explores a different approach: the use of virtualization to bridge the gap between SMM systems and workstation clusters. We present vNUMA (“virtual NUMA”), a virtual shared-memory multiprocessor built from a cluster of commodity workstations. A hypervisor runs on each node of the cluster and manages the physical resources. A single virtualized instance of an OS, such as Linux, is then started on the cluster. This OS and its applications executes on a virtual ccNUMA machine with many virtual CPUs. The virtualization layer transparently maps the virtual CPUs to real CPUs in the cluster, and provides DSM using software techniques. In this way, a single OS instance can be scaled “outside the box”, utilizing the computing resources of more than one node. Users gain all of the advantages of such an SSI multiprocessor, such as a single view of resources and transparent process scheduling.

The core ideas of vNUMA have been presented in an earlier short paper [7]. Here we focus on the design and implementation issues that are critical to making vNUMA work. We address the problem of constructing a high-performance virtual NUMA system on commodity hardware by:

- an approach to write sharing which individually intercepts sparse write accesses, while falling back to a page-based write-invalidate protocol when appropriate,
- introducing the technique of write-broadcast with deterministic incremental merge for providing total store order, and
- demonstrating an efficient approach for avoidance of page thrashing.

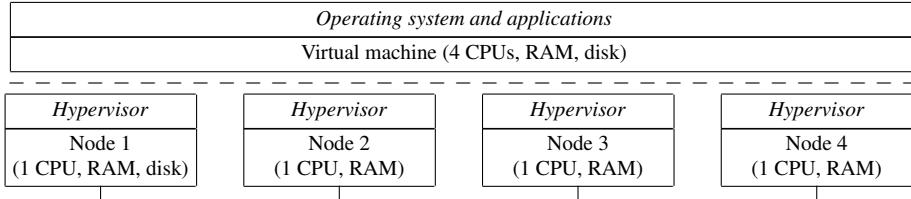


Figure 1: Example vNUMA system

In the next section we present an overview of the vNUMA hypervisor and its DSM system, which is designed for a small cluster of commodity workstations. In Section 3 we discuss a number of enhancements to established DSM protocols that improve their suitability for use inside a hypervisor. Section 4 takes a detailed look at implementation issues, including architecture-specific optimisations. Section 5 presents an evaluation of our vNUMA prototype. Related work is summarised in Section 6.

2 vNUMA Overview

2.1 Approach

In order to minimise overheads, vNUMA is designed as a Type-I hypervisor, executing on bare hardware with no host OS. Our prototype was built on Itanium workstations, which are frequently deployed in clusters for high-performance computing (HPC) use. While the vNUMA design is independent of a specific ISA, the implementation does use processor-specific optimisations.

The majority of previous software DSM systems have been designed as middleware running on top of an OS. In vNUMA, the DSM system is integrated with the hypervisor. There are two levels of memory address translation in a virtualized system. The guest OS maps applications’ virtual addresses onto a *guest-physical* address space, which represents the physical memory of the virtual machine. Then, the hypervisor maps guest physical addresses to real physical addresses on a host computer. This lower layer, transparent to the guest OS, is where the vNUMA DSM system operates. It provides operating systems with the illusion of a single physical address space across multiple host computers, as indicated in Figure 1.

As a result, the shared address space in vNUMA comprises not just some subset of data memory that is known to be shared, but all of the memory of the virtual machine. Since our aim is to run unmodified application binaries (and, ideally, unmodified OSes), vNUMA must faithfully reproduce the hardware SMP programming model. Doing this efficiently presents challenges. On the other hand, vNUMA runs in the processor’s privileged mode, which gives it access to certain techniques

that may be difficult or prohibitively inefficient for a userspace DSM system. Examples include the efficient emulation of individual instructions, and the use of the performance-monitoring unit (PMU) to track the execution of specific instructions.

2.2 Basic DSM protocol

At the heart of the vNUMA DSM system is a simple single-writer/multiple-reader write-invalidate protocol based on the Ivy protocol [23]. For page location, vNUMA implements a fixed distributed manager scheme, whereby the global guest-physical address space is divided into equal-sized portions; each node acts as a *manager* for one of these portions.

vNUMA’s transparency requirements imply that the concept of a manager node is unknown outside the hypervisor. However, efficiency is improved if the guest OS has a notion of locality. vNUMA uses the concept of NUMA node-local memory to ensure that the guest will favour locally-managed memory when making allocation decisions, and as such works best with a NUMA-aware guest OS. While for normal DSM systems the concept of the manager node is a complication required for efficiency, for the virtual NUMA system it is actually a good match.

vNUMA’s DSM algorithm is based on the a version of the Ivy protocol which the Ivy authors describe as the “improved” protocol. The improvement keeps the *copyset* information (where copies of a page are held) with a changing page *owner* rather than the manager. This helps to minimise the number of messages required, and to avoid deadlock issues that are a problem with the basic protocol [13].

3 Enhancements to DSM Protocols

Latency of DSM operations is the crucial limiting factor for the performance of vNUMA. Whenever a fetch or invalidation message is sent, consistency requires that execution on the local processor must stall until the response is received. Here we discuss protocol improvements that are designed to minimise the number of stalls and messages required for DSM operation.

3.1 Double faults and ownership

In the original Ivy protocol, a page that has been fetched on a read fault would have to be re-fetched on a subsequent write fault in order to ensure consistency. A later optimisation avoided the double transfer with the help of version numbers [20]. We use an optimisation that seems to have been used in Mirage [11]: an owner can determine whether the page data needs to be sent simply by consulting the page’s copyset information. This is because any intervening writes would have invalidated the faulting node’s read copy and hence removed it from the copyset.

Another optimisation also goes back to Mirage but is simplified in vNUMA: as soon as the manager becomes a member of the copyset, ownership is automatically transferred to the manager (Mirage required extra messages for this).

3.2 Addressing sparse data accesses

Minimising the number of communication events in a distributed shared memory system depends critically on caching of remote data. Many commonly used data structures, such as linked lists and trees, tend to have poor spatial locality, and may result in a processor accessing many pages. If locally cached copies of these pages can be accessed, then overheads are small, but if each of the pages regularly requires a remote fetch, performance will suffer greatly.

In the absence of writes, pages eventually become read-shared, allowing each processor to access the cached copy of those pages without any communication. This is clearly desirable. Now consider that some processor occasionally writes a value to a certain page that is otherwise read-shared. In the Ivy protocol, first the writer must stall while all of the read copies are invalidated, then all of the active readers eventually stall and re-fetch the entire page data. Clearly it would be more efficient, for such sparse updates, to propagate the individual write to any readers.

3.2.1 Write detection

In any such protocol, writes must be detected and write update messages sent to other nodes. Write detection at sub-page granularity is a challenge to implement efficiently. *Page diffing*, as implemented in Munin [3] and many later systems, cannot be used by vNUMA, for several reasons.

Firstly, by the time that the diffing is performed, information has been lost about the size of the writes, which has implications for the outcome of conflicting writes. For example, assume that a 4-byte integer variable has an initial value of 0. Consider a case where processor P1 writes 1 to the variable, P2 writes -1, and then P3 issues a read. The Itanium architecture dictates

that the outcome will be one of 0, -1 or 1 (depending on which of the writes have been seen at P3). However, the diff generated at P1 may contain as little as one byte, since in binary representation only one byte of the value has changed. The diff generated at P2 contains four bytes, since all four bytes of the binary representation have changed (-1 = 0xffffffff in hexadecimal). After both diffs are applied, the value at P1 may be 0xfffffff01, which is not one of the valid outcomes. Differing at a 32-bit granularity would solve this problem for 32-bit values, but there would still be problems with smaller and larger types. Systems that employ differing, such as TreadMarks [19], rely on the programmer to avoid issuing conflicting writes within an interval, and to take care when using smaller types than the diff granularity. However, at the ISA level there is no such requirement; in fact the example above is completely legal if the programmer does not require a guarantee as to which change is applied first. This would present problems for legacy code on vNUMA.

Secondly, the standard differing approach involves making the page freely writable on the first write access, in order to avoid further write faults. However, if a page is both readable and writeable, then atomic read-modify-write instructions such as compare-and-exchange will freely execute, thus destroying their semantics. User-level DSM systems that employ differing schemes can avoid this issue by stating that the programmer must use the synchronisation constructs provided by the DSM system, and not rely on the behaviour of atomic instructions to shared memory. This is not practical for vNUMA.

An alternate approach, *software write detection*, as used in Midway [37], relies on compiler support. This would prevent transparent distribution of legacy applications, and is therefore also not suitable for vNUMA.

We therefore attempt to intercept writes individually, a technique we describe as *write trapping*. While this is prohibitively expensive for user-level DSM systems, the overhead can be kept much smaller in a thin hypervisor such as vNUMA. The current C language implementation results in an overhead of around 250 cycles per write, but this is largely due to compiler limitations; in theory under 100 cycles should be achievable.

Even so, writes are frequent operations and trapping every write in the system would be impractical; indeed the majority of pages in the system are not actively write-shared at all. vNUMA uses an adaptive scheme which changes a page’s mode between this write-trapping (write-update) mode and the basic write-invalidate mode, depending on the access pattern.

The adaptation scheme currently implemented is similar to the *read-write-broadcast* (RWB) protocol [31] developed for hardware cache coherence. The run-length

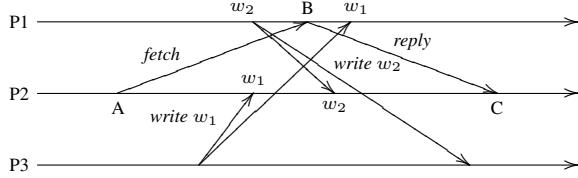


Figure 2: Timeline showing a possible ordering problem

of local writes to a page that are uninterrupted by writes received from other nodes is tracked with a counter. If the count exceeds a threshold, trapping of individual writes ceases and the page is transitioned to write-invalidate mode, in which we use the conventional Ivy-like write-invalidate protocol described earlier. This can reflect two types of access patterns — either one node is accessing the page exclusively, or one node is making a large number of updates to the page in a short time — and in both cases invalidation is likely to perform better. The decision is made individually by each node, so even if one node chooses to acquire the page exclusively, other infrequent writers continue to intercept writes to the page and report them back to the exclusive owner (providing there are no reads).

This scheme makes its decision purely on the basis of tracking write accesses. Its drawback is that it will not detect producer-consumer sharing with a single intermittent writer and multiple readers. This leads to periodic invalidation of the readers’ copies and subsequent re-faulting, even though the write-update mode may be better in this case. An improved algorithm might be one similar to the *efficient distributed write protocol* (EDWP) [1], which tracks both read and write accesses, and prevents a transition to exclusive mode if more than one processor is accessing the page. However, this is considerably more complex (since sampling read accesses is required) and has not been implemented.

3.2.2 Write propagation

For pages in write-update mode, vNUMA broadcasts writes to all nodes. While this may seem inefficient, it has some advantages; it greatly reduces the complexity of the system and naturally results in total store order (TSO) consistency. Per-packet overheads are amortized by batching many writes into a single message (see Section 4.3). Certainly this design choice would limit scalability, but vNUMA is designed for optimal performance on a small cluster.

Each node generally applies any write updates that apply to pages that it has read copies of, and discards any irrelevant updates. However, care must be taken when applying write updates to a page that is being migrated. A node P2 receiving a page from P1 queues the updates

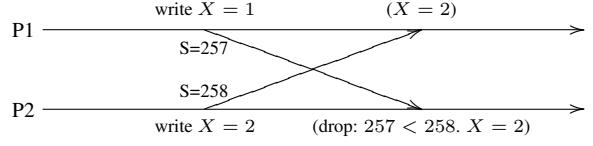


Figure 3: Coherence problem with write notices, and its resolution by deterministic merging according to sequence numbers.

it receives while the page is in flight. Then, it must apply the subset of queued writes that have not already been applied at P1. In other words, P2 must apply exactly those updates which were received at P1 *after* P1 sent the page to P2. An example is shown in Figure 2: write w_1 must be applied, while w_2 must be discarded.

Our algorithm for determining which writes to apply assumes that the network provides *causal order delivery*, which is a property of typical Ethernet switches (c.f. Section 4.5). We provide a brief description here, more details are available elsewhere [6].

We maintain at each node a counter of writes, and that counter value is included in a page-fetch reply message. As per Figure 2, A denotes the event of P2 sending a fetch message to P1, B the event of P1 receiving that message and immediately replying to P2, and C the event of P2 receiving the page. In the figure, the respective counter values are $N_A = 0$, $N_B = 1$, and $N_C = 2$. N_1 denotes the number of writes from P1 queued at P2 at event C ($N_1 = 1$ in the figure). The algorithm then becomes:

- discard the N_1 messages pending from P1;
- out of the remaining writes, apply the latest $N_C - N_B$ (and thus discard the earliest $N_B - N_A - N_1$ writes).

In the example, the first step will drop w_2 and the second step will apply w_1 .

3.2.3 Deterministic incremental merge

The write-update algorithm as presented so far is insufficient to guarantee coherence in a strict sense. In the example shown in Figure 3, where nodes P1 and P2 simultaneously write to a location X , P1 could observe $X = 1$ followed by $X = 2$ while P2 observes $X = 2$ followed by $X = 1$, in violation of coherence. Two solutions to this problem exist in the literature [8]: a central sequencer or associating every write with a globally-unique sequence number. The central sequencer, while guaranteeing that all nodes converge on the same value, does not prevent intermediate values from being observed at a single node, in violation of the architecture’s specification of memory coherence. It also presents a bottleneck.

A globally-unique sequence number can be implemented as a local sequence number — synchronised on communication — with the node number as a tie-breaker where no causality relationship exists [8, 21]. However, the conventional deterministic merging approach [8] would involve waiting to receive write messages from all nodes before deciding on a final value. As vNUMA only sends write messages as needed, a particular node may be quiet for a considerable time, which would necessitate regular empty write messages to ensure coherence.

Note, however, that coherence only requires total ordering on a per-location basis. Consider the case where $\{w_1, w_2, \dots, w_n\}$ are a set of writes to the same location, ordered by their global sequence number. From the point of view of program semantics, it is not essential to guarantee that all of $\{w_1..w_n\}$ are observed at any particular node, as long as the observed subset follows the correct ordering and culminates in the proper final value. In other words, observing $\{w_2, w_1, w_n\}$ is not allowed since w_1 must precede w_2 , but observing $\{w_1, w_n\}$ or even just $\{w_n\}$ is allowable. Omitted intermediate values could correspond to the case where a processor was not fast enough to observe the intervening values.

We make use of this fact to implement a technique we call *incremental deterministic merging*. Each incoming write notice is applied immediately, but it is only applied to a certain location if its sequence number is greater than that of the last write to that location. Since every node receives all write notices, the value of that location always ultimately converges on the write with the maximum sequence number (w_n), with any intermediate values respecting the required ordering. Figure 3 shows how this resolves the original problem.

3.3 Atomic operations

The protocol described so far is sufficient for correctness, but highly inefficient for hosting an OS (such as Linux) that uses atomic instructions (xchg, fetcchadd or cmpxchg) to implement kernel locks. Any of those operations results in a fall-back to write-invalidate mode, making kernel locks very expensive. We therefore introduce an extension to the protocol, which we call *write-update-plus* (WU+).

An important observation is that, in the Itanium architecture and other typical processor architectures, there is no requirement for ordering between an atomic read-and-write instruction and remote reads. A remote read can safely return either the value before or after the atomic operation. Thus, there is no need for invalidation of read-only copies when an atomic operation is encountered; the write phase of the operation can be propagated to readers via the write-update mechanism.

However, in order to guarantee atomicity of the read

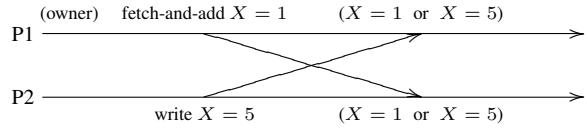


Figure 4: Simultaneous atomic operation and remote write. P1 is the owner of X and therefore has permission to execute atomic operations. According to the Itanium architecture, the correct result is either 5 or 6, depending on which operation appears first in the total order. Here, even with deterministic merging, $X = 1$ may occur.

and write phases, only one processor at any time can be allowed to perform an atomic operation to a particular location. In the WU+ protocol, we enforce that only the owner of a page can execute atomic operations on that page. Any other node must first acquire ownership.

In addition, simultaneous atomic operations and remote writes can lead to incorrect results, as shown in Figure 4. The WU+ protocol therefore enforces a single writer for pages targeted by atomic operations. Thus, at any point, a page can be in one of three modes: *write-invalidate*, *write-update/multiple-writer*, or *write-update/single-writer*. The transition from multiple- to single-writer mode occurs when atomic operations to a page are intercepted; nodes are synchronously notified that they can no longer generate write updates to the page without acquiring ownership.

4 Implementation

The implementation of vNUMA is around 10,000 lines of code. Of this around 4,000 lines constitute a generic Itanium virtual machine monitor, the DSM system is around 3,000 lines, and the remainder deals with machine-specific initialisation and fault handling. In total the hypervisor code segment is about 450KiB (Itanium is notorious for low code density).

Besides generic protocol optimisations, we used a number of implementation techniques to optimise performance, which we discuss in this section. Some of these are processor-independent, others make use of particular Itanium features (but similar optimisations can be made for other ISAs).

4.1 Avoiding thrashing

A naïve DSM implementation suffers from a page thrashing problem, indicated in Figure 5. If two nodes simultaneously write to a page, the page may be transferred back and forth with no useful work done. A frequently-used solution to this problem is to introduce an artificial delay to break the livelock. However, this is non-optimal by design, as there is no easy way to determine an appropriate delay, and the approach increases

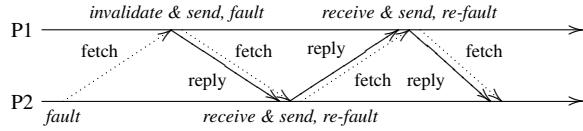


Figure 5: Timeline demonstrating the page thrashing problem. Solid lines indicate transfers of ownership.

latency. Instead, we use an approach that guarantees that at least one instruction is executed before a page is transferred.

One way to implement this is by putting the machine into single-step mode after receipt of a page, and not processing any page requests until the trap that is caused by the execution of the next instruction is processed (at which time normal execution mode is resumed).

A cheaper alternative (implemented in vNUMA) is to consult the performance-monitor register that counts retired instructions to determine whether progress has been made since the last page transfer. (Note that checking the instruction pointer is not sufficient, as the code might be executing a tight loop, which could mask progress.) If lack of progress is detected, then one could fall back to the single-step approach. Instead we optimistically continue and re-check after a short delay. While this is similar to the timed-backoff scheme implemented in other DSM systems, vNUMA can use a very short delay to minimise latency, as the hypervisor can prevent preemption and thus ensure the opportunity for progress.

A complication of the chosen scheme is that one instruction may access several pages, up to four on the Itanium (an instruction page, a data page and two register-stack engine pages). This introduces the possibility of a circular wait condition, and thus deadlock.

We prevent deadlock by applying the anti-livelock algorithm only to pages accessed via explicit data references, and not instruction or register stack pages. Since the data reference is always logically the last reference made by an instruction — occurring after the instruction reference, and after any register stack accesses — instruction completion is guaranteed once the data page is obtained, and there is no possibility of deadlock. Indeed it is not necessary to apply the livelock prevention algorithm for instruction and register stack references, since instruction accesses are always reads, and the Itanium architecture specifies that register-stack pages should not be simultaneously accessed by multiple CPUs (or undefined processor behaviour could result). Even if a malicious application were to invoke this livelock case, it would not prevent the operating system from taking control and the process could be killed. Thus, this strategy prevents livelock in a well-behaved operating sys-

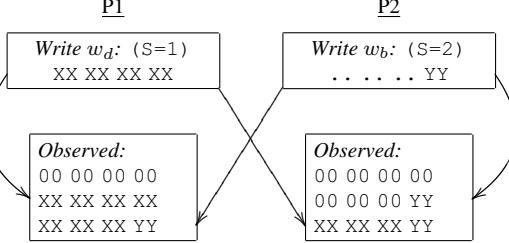


Figure 6: Combining writes of different sizes. On P2, write w_d appears to modify 3 bytes.

tem while also preventing any possibility of deadlock.

On some other architectures such as x86, this approach might still result in deadlock, since a single instruction may access several data pages. One possibility would be to release pages after a random period of time, even if no progress is made. In the worst case, this reintroduces the problems associated with backoff algorithms, but should perform better in the common case, while ensuring that a permanent deadlock does not occur.

4.2 Incremental merging

In Section 3.2.3 we somewhat vaguely referred to “locations” as the destinations of writes. Given that real architectures support writes of different sizes, we need to understand at which granularity conflict resolution must be applied. Figure 6 demonstrates that it must be applied at the byte, not the word level: the 4-byte write w_d at P1 with sequence number $S(w_d) = 1$ logically precedes the byte-sized write w_b at P2 with $S(w_b) = 2$. If the newer byte-sized write happens to be applied first at some node, then when the older 4-byte write is received, it must only appear to modify the top 3 bytes. This set of observed values is consistent with the Itanium memory consistency model [16].

This makes efficient implementation a challenge, as keeping separate sequence numbers for each byte of memory is clearly prohibitive. As the majority of updates do not conflict, tracking overhead must be minimised.

Fortunately, sequence-number information only needs to be kept for short periods. Once updates with a certain minimum sequence number are received from all nodes, all information related to lower sequence numbers can be discarded.

This observation enables an implementation of sequence numbers that is simple and has low overheads. We use a fixed-size buffer that stores information about a certain number of preceding writes (Figure 7). Each write is described by the address of the 64-bit machine word that it targets and a mask of bytes within that word (note that we assume that writes never cross a machine-

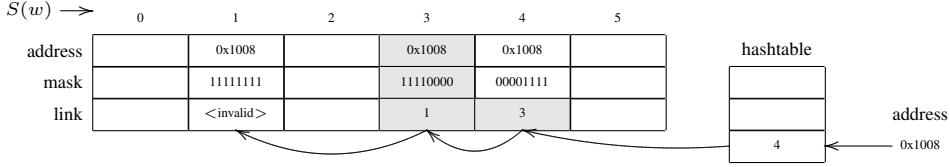


Figure 7: Data structure for coherence algorithm. The example shows an incoming write with sequence number 3, address 0x1008 and mask 11111111 (entire 8 bytes); the unshaded fields show the “before” state (but note that entry 4 is originally linked to entry 1). The hash chain is traversed as far back as sequence number 4; since that logically newer write wrote 00001111 (the lower four bytes), the mask is constrained to 11110000 (the top four bytes). The appropriate slot for the new write is then updated and linked in place.

word boundary). Writes are directly inserted into the buffer using the least significant bits of their sequence number as an index; assuming that sequence numbers are allocated in a unique and relatively dense fashion, this mapping is quite efficient. For fast lookup, writes are then indexed using a hash function of their target address; writes with the same hash value are linked together in a chain. This chain is always kept in reverse sequence number order.

The only operation on this data structure is adding a new write. While traversing the linked list to insert a write, all logically newer writes to the same address are encountered, which are used to constrain the mask of bytes to be written. Once a link field with an older sequence number is reached, traversal stops and the new write is inserted into the chain. The constrained mask is returned and used to determine the bytes in memory that are actually modified.

Since a chain is never traversed past the sequence number of a newly received write, the chains need never be garbage-collected. It is sufficient to make the buffer large enough so that it covers the window of sequence numbers that can be received from other nodes at any time. Since each node tracks the last sequence number received from each other node, a violation of this rule can be detected and a stall induced if necessary; however such stalls are clearly undesirable and can be eliminated by ensuring that each node does periodically send updates.

4.3 Write batching

Write update messages are small, and vNUMA batches many of them into a single Ethernet message in order to improve performance. Batching can make use of the processor’s weak memory ordering model. The Itanium architecture uses *release consistency*: normal load and store instructions carry no ordering guarantees, but load instructions can optionally be given acquire semantics (guaranteeing that they become visible prior to subsequent accesses), while store instructions can optionally have release semantics (guaranteeing that they become

visible after preceding accesses).

Acquire semantics require no special care, since the processor guarantees this behaviour on local operations, and because operations are never visible remotely before they are visible locally.

Release semantics require special care, however. Consider an access A that is followed by a write with release semantics W_{rel} . A must become visible on all nodes before W_{rel} . The processor interprets the release annotation and guarantees that A completes before W_{rel} . However, in the case that A is a write, local completion does not imply remote visibility — writes may be queued by vNUMA before being propagated to remote nodes. It is up to vNUMA to guarantee that A is observed before W_{rel} .

This is trivial if W_{rel} is to a write-update page: if A is to an exclusive page, it becomes visible immediately and thus necessarily before W_{rel} ; if not, then the DSM system simply needs to ensure that the writes are sent in order. The interesting case is where W_{rel} is to an exclusive page and A is a queued write to a write-update page. In this case, the DSM system needs to ensure that W_{rel} is propagated before a read response to A .

The challenge is to detect when W_{rel} is to an exclusively-held page, as this cannot be made to trap without making all ordinary writes to the same page fault as well. Fortunately, the Itanium performance monitoring unit (PMU) provides a counter which can be configured to count releases. When a read request arrives for an exclusive page, the counter is checked to determine whether a release occurred on the last interval. If so, the write buffers are flushed before sending the read response.

As an additional optimisation, the write queue is eagerly flushed at the time that a write is intercepted, if a release has been seen (either on that instruction or in the previous interval) and if the network card transmit queue is empty. This expedites transmission of writes, since a release is usually used in the context of data that is intended to be observed by another processor. If the transmit queue is not empty, then the flush is scheduled

to occur after a delay; this rate-limits the update packets and allows additional writes to accrue while previous update packets are being transmitted.

4.4 Memory fences

Itanium also provides a memory fence instruction, `mf`, that has both acquire and release semantics: loads and stores cannot pass it in either direction. The PMU counts `mf` as a release (as well as an acquire), so the above detection mechanism can be used to ensure that writes are ordered correctly across a fence. The one case that is problematic is the ordering between writes and subsequent reads. If a write is separated from a subsequent read by a fence, as in Figure 8, then the strict semantics of `mf` would require preventing the read from returning a cached copy before the write is visible everywhere. In practice this means that if both both X and Y are initially zero, at most one processor is allowed to read that value.



Figure 8: The memory fences prevent that both processors’ reads return the initial values of the respective variables.

A strict implementation of the `mf` semantics would have severe performance implications in vNUMA. Instead, we decided to compromise our goal of full transparency, and require that `mf` operations are replaced by atomic operations (equivalent to a lock-based implementation of `mf`). Despite the assortment of synchronisation algorithms implemented in Linux, only one case was encountered in testing which required a full fence — the implementation of the `wait_on_bit_lock` function — and this was resolved via a simple modification.

4.5 Inter-node communication

vNUMA performance is highly sensitive to communication latency. This rules out hosting device drivers inside a guest OS as done in many modern virtual-machine monitors. Instead, vNUMA contains, inside the hypervisor, latency-optimised drivers for a number of Gigabit Ethernet chipsets.

We further minimise communication overhead by defining a very simple protocol at the Ethernet layer. We use the coalescing feature of Ethernet cards to separate the headers and payload into different buffers to enable zero-copy in the common case (in the special case where a local write occurs while a page is being sent, a shadow copy is created). Transfers of 4KiB pages either use a single ‘jumbo’ frame or are broken into four fragments.

Fragmenting the packet is actually preferable to reduce latency, since the fragments can be pipelined through the network (this is also why four fragments are preferable to three, although above this the overheads outweigh the benefits).

vNUMA also makes extensive use of known properties of networking hardware, in order to avoid protocol overhead where possible. Specifically, vNUMA relies on the network to be (mostly) *reliable*, to provide *causally-ordered delivery*, and ideally to provide *sender-oblivious total-order broadcast*. The last requirement means that if P_1 broadcasts m_1 , and P_2 broadcasts m_2 , then either all other observers observe m_1 before m_2 , or all other observers observe m_2 before m_1 . “Sender-oblivious” means that P_1 and P_2 do not need to make any conclusions about the total order; this is an optimisation geared towards Ethernet, where a sender does not receive its own broadcast.

Causally-ordered delivery is guaranteed by the design of typical Ethernet switches. Reliability is not guaranteed, but packet loss is very rare. vNUMA is therefore optimised for lossless transmission. Timeouts and sequence numbers, combined with a knowledge that the number of messages in-flight is bounded, are used to deal with occasional packet loss.

Total-order broadcast usually holds in small switches but may be violated by a switch that contains several switch chips connected by a trunk, as a broadcast will be queued in a local port on one chip before forwarded over the trunk. It may also be violated when packets are lost. In this case, remote store atomicity may not hold in vNUMA. This could potentially be resolved with a more complex protocol for store atomicity, similar to our approach to coherence. We did not design such a protocol. In practice, this limitation is of little significance; many other processor architectures including x86 also do not guarantee store atomicity.

4.6 I/O

vNUMA contains support for three classes of virtual devices: network (Ethernet), disk (SCSI) and console.

The **network** is presented as a single virtual Ethernet card. As processes arbitrarily and transparently migrate between nodes, and TCP/IP connections are fixed to a certain IP address, transparency requires a single IP address for the cluster. Outgoing messages can be sent from any node, vNUMA simply substitutes the Ethernet address of the real local network card into outgoing packets. Incoming packets are all received by a single node. This has the advantage that the receiving part of the driver and network stack always runs on a single node, but the disadvantage that the actual consumer of the data may well be running on a different node.

The ideal approach for dealing with **disks** would be

to connect them to a storage area network (SAN), so that they can be accessed from any of the nodes. This is done by Virtual Iron’s VFe hypervisor [34], but is in conflict with vNUMA’s objective of employing commodity hardware. Therefore, the vNUMA virtual machine provides a single virtual SCSI disk. The present implementation routes all disk I/O to the bootstrap node, which contains the physical disk(s). It would be possible to remove this bottleneck by striping or mirroring across available disks on other nodes.

The **console** is only supported for debugging, as users are expected to access the vNUMA system via the network. All console output is currently sent to the local console (which changes as processes migrate). Input can be accepted at any node.

4.7 Other implementation issues

vNUMA virtualizes inter-processor interrupts (IPIs) and global TLB-purge instructions in the obvious way, by routing them to the appropriate nodes.

In order to boot up a vNUMA system, all of the nodes in the cluster must be configured to boot the vNUMA hypervisor image in place of an operating system kernel. Then, one of the nodes is selected by the administrator to be the bootstrap node, by providing it with a guest kernel image and boot parameters; the other nodes need no special configuration.

Once the bootstrap node initialises, it uses a discovery protocol to find the other nodes and their resources, and provides them with information about the rest of the cluster. It then starts executing the guest kernel. As part of its normal boot process, the guest OS registers an SMP startup address and wakes the other nodes by sending IPIs. The other nodes start executing at the given address in the globally-shared guest-physical address space, thus faulting in the OS image on demand.

4.8 Limitations

Like the ubiquitous x86 architecture, Itanium was originally not trap-and-emulate virtualizable [24]. While this has now been mostly remedied by the VT-i extensions [17], a number of challenges [14] remain, particularly relating to the register stack engine and its interaction with the processor’s complex translation modes. vNUMA utilizes some para-virtualization of the guest OS, and thus presently only supports Linux guests.

5 Evaluation

We evaluated vNUMA using three types of applications, which cover some of the most common use scenarios for large computer systems: computationally-intensive scientific workloads, software-build workloads, and database server workloads.

5.1 Test environment

Our test cluster consisted of eight HP rx2600 servers with 900MHz Itanium 2 processors, connected using a Gigabit Ethernet via an HP ProCurve 2708 switch. Since vNUMA does not yet support SMP within a node, only one CPU was used in each server.

The guest OS was Linux 2.6.16, using default configuration settings where possible, including a 16KiB page size. An exception are the Treadmarks measurements, which were performed with a 4KiB page size to provide a fair comparison of DSM performance (since vNUMA subdivides pages to 4KiB granularity internally).

Pre-virtualization [22] was used to automatically transform the Linux kernel for execution on vNUMA (our Itanium machines are not VT-i enabled). Three minor changes were made manually. Firstly, the Linux `wait_on_bit_lock` function was modified as described in Section 4.4. Secondly, the `clear_page` function was replaced with a hypervisor call to allow it to be implemented more optimally. Finally, the kernel linker script was modified to place the `.data.read_mostly` section on a separate page to ease read-sharing (the default setup co-allocates this section with one which contains locks).

Results presented are a median of the results from at least ten runs of a benchmark. The median was chosen as it naturally avoids counting outliers.

5.2 HPC benchmarks

HPC is a main application of compute clusters, and therefore a natural application domain for vNUMA. While many HPC applications use an explicit message-passing paradigm as supported by libraries such as MPI [26], a significant number rely on hardware-supported shared memory or DSM, and are therefore well-suited to execution on vNUMA. We used TreadMarks [19] as a DSM baseline. While TreadMarks may no longer represent the state of the art in DSM research, it is one of the few DSM systems that has been widely used in the scientific community.

TreadMarks is distributed with an assortment of benchmark applications, mostly from the Stanford SPLASH-2 suite [36] and the NAS Parallel Benchmarks from NASA [10]. To avoid biasing the evaluation against TreadMarks, we used the unmodified TreadMarks-optimised sources, and for vNUMA provided a stub library that maps TreadMarks APIs to `fork()` and shared memory. We also ran the benchmarks on one of our SMP servers on native Linux to show best-case scalability (although limited to the two CPUs available).

Figure 9 shows an overview of results for each benchmark. While the ultimate limits of scalability are difficult to establish without a much larger cluster, vNUMA

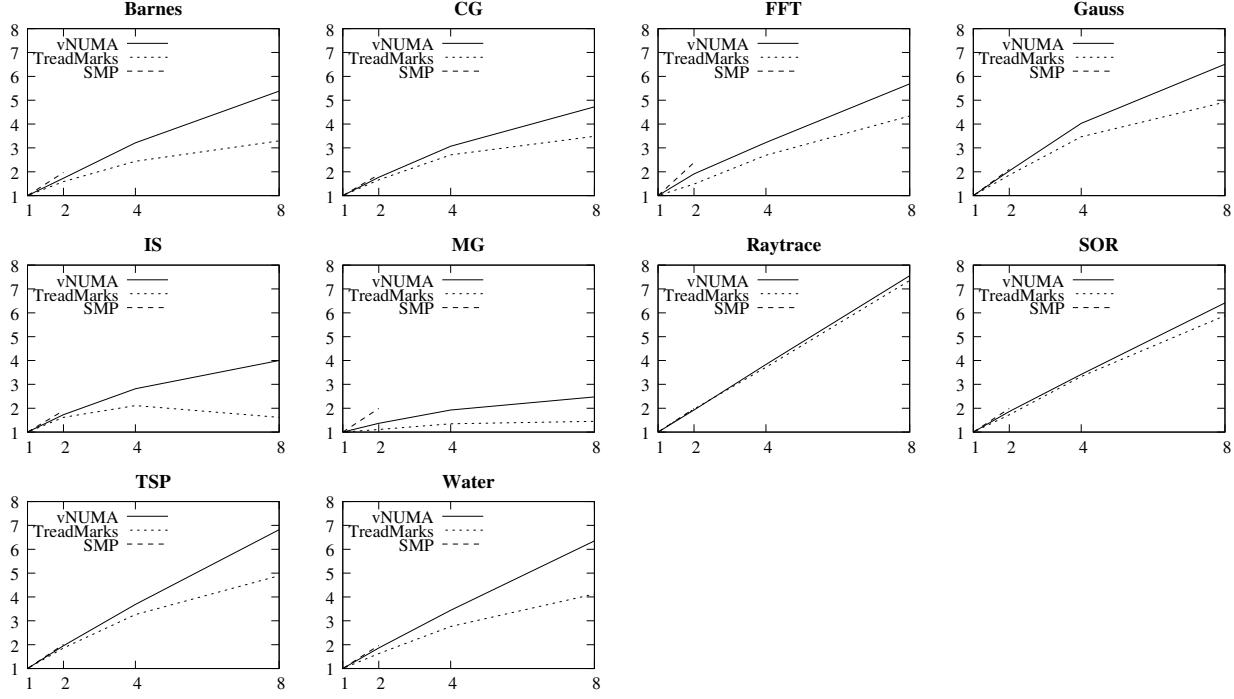


Figure 9: HPC benchmark performance summary. Horizontal axes represent number of nodes, vertical axes represent speed-up.

was designed for optimal performance on a small cluster. As the graphs show, vNUMA scalability is at least as good as TreadMarks on all benchmarks, and significantly better on **Barnes**, **Water**, **TSP** and **IS**. In absolute terms **MG** exhibits the poorest scalability, but it is a benchmark that poses challenges for all DSM systems, due to the highly irregular sizes of its three-dimensional arrays.

5.3 Compile benchmark

Large servers and clusters are frequently used for software builds. Figure 10 compares vNUMA’s scalability with `distcc` [29] when compiling vNUMA. As compilation throughput tends to be significantly affected by disk performance, we eliminated this factor by building on a memory file system (RAM disk).

The figure shows that vNUMA scales almost exactly as well as `distcc`. The line labelled “Optimal” is an extrapolation of SMP results, based on an idealised model where the parallelisable portion of the workload (86 %) scales perfectly. On 4 nodes, the ideal speed-up is 2.8, while both vNUMA and `distcc` achieve 2.3. On 8 nodes, the ideal speed-up is 4.0, while both vNUMA and `distcc` achieve 3.1.

In the case of `distcc`, the overheads stem from the centralised pre-processing of source files (which creates a bottleneck on the first node), as well as the obvious overheads of transferring source files and results over the

network. In the case of vNUMA, the largest overhead is naturally the DSM system. Of the 15 % overhead accountable to vNUMA in the four node case, DSM stalls comprise 7 %, the cost of intercepting writes is around 3 %, network interrupt processing around 2 % and other virtualization overheads also around 2 % (see also Section 5.4).

The majority of the DSM stalls originate from the guest kernel. This is because the compiler processes do not themselves communicate through shared memory. Their code pages are easily replicated throughout the cluster and their data pages become locally owned. However, inputs and outputs are read from and written to the file system, which shifts the burden of communication onto the kernel. In general, the compile benchmark can be considered representative of an application that consists of many processes which do not interact directly

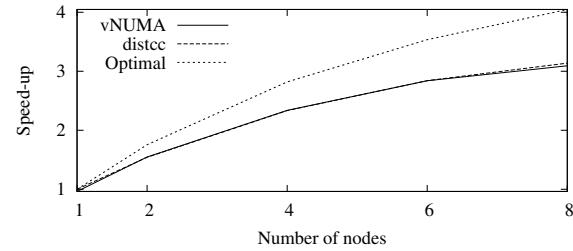


Figure 10: Compile benchmark performance summary

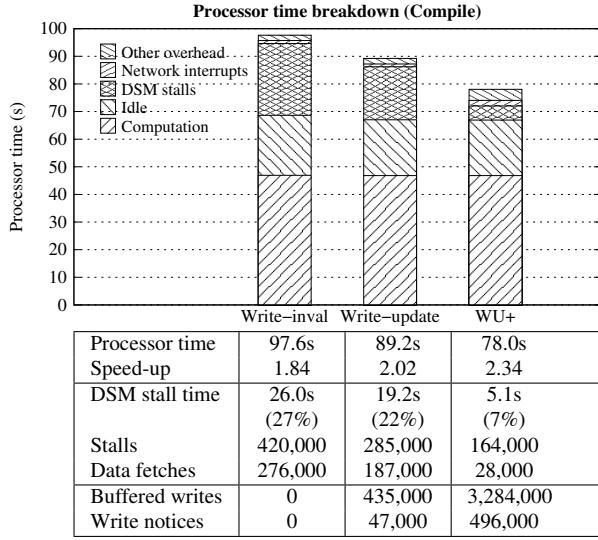


Figure 11: Effect of protocol on compile benchmark

but interact through the filesystem.

Profiling the kernel overheads shows that the largest communication costs arise from maintaining the page cache (where cached file data is stored), and acquiring related locks. Similarly the file system directory entry cache (which caches filenames), and related locks, also feature as major contributors. Nonetheless, considering that the overall overhead is no greater than that of `distcc` — a solution specifically crafted for distributed compilation — this seems a small price to pay for the benefits of a single system image.

5.4 Effect of DSM protocol optimisations

To quantify the benefits of the chosen DSM protocols, we also executed the compile benchmark at three different levels of protocol optimisations: using the basic Ivy-like write-invalidate protocol, using our write-update protocol, and using our write-update-plus (WU+) protocol which intercepts atomic operations as well as ordinary writes. The results are summarised in Figure 11.

Performance is improved significantly by the more advanced protocols, with speed-up on four nodes increasing from 1.84 to 2.02 to 2.34. This is due to a sharp reduction in the number and latency of stalls. With the write-invalidate protocol, 420,000 synchronous stalls are incurred, totalling 26.0 seconds (an average of 62 μ s/stall, which is dominated by the high latency of fetching page data that is required in 66% of cases). The write-update protocol reduces the number of synchronous stalls to 285,000, with a proportional decrease in stall time to 19.2s. However, the write-update-plus protocol has the most dramatic impact, reducing stall time to only 5.1s. While the total number of stalls is still

164,000, the majority of these are now ownership transfers, which involve minimum-length packets and therefore have low latency (17 μ s in the common case). The number of stalls that must fetch data has decreased to only 28,000, which shows the effectiveness of this protocol in enhancing read-caching.

The price of this improved read-caching is that many more writes must be intercepted and propagated, which is reflected in higher overheads both for intercepting the writes (reflected in hypervisor overhead) and at the receivers of the write notices (reflected in interrupt overhead). Nonetheless there is still a significant net performance improvement.

5.5 Database benchmark

Databases present a third domain where high-end servers and clusters are used. We benchmarked PostgreSQL [30], one of the two most popular open source database servers used on Linux. The open-source nature was important to be able to understand performance problems. For the same reason — ease of understanding — simple synthetic benchmarks were employed instead of a complex hybrid workload such as TPC-C. Two tables were initialised with 10,000 rows each: one describing hypothetical users of a system, and the other representing posts made by those users on a bulletin board. A pool of client threads then performed continuous queries on these tables. The total number of queries completed in 30 seconds (after 5 seconds of warm-up) is recorded. This is similar in principle to benchmarks like TPC-C, but utilizes a smaller number of tables and a simpler mix of transactions.

Four different types of queries were used: **SELECT**, which retrieves a row from the users table by matching on the primary key; **SEARCH**, which retrieves a row from the users table by searching a column that is not indexed; **AGGREGATE**, which sums all entries in a certain column of the users table, and **COMPLEX**, which returns information about the five most prolific posters (this involves aggregating data in the posts table, and then performing a ‘join’ with the user table).

The results are summarised in Figure 12. vNUMA performs well for **COMPLEX**, which involves a base throughput of tens of queries a second. However, performance is degraded for the higher-throughput workloads, **SEARCH** and **AGGREGATE**, and most significantly so for **SELECT**, which involves little computation per query and can thus usually achieve thousands of queries a second on a single node. **SEARCH** and **AGGREGATE** barely manage to regain single-node performance on 8 nodes, while **SELECT** does not scale at all.

The cause of this throughput-limiting behaviour is simple: using multiple distributed nodes suddenly introduces the potential for much larger communication

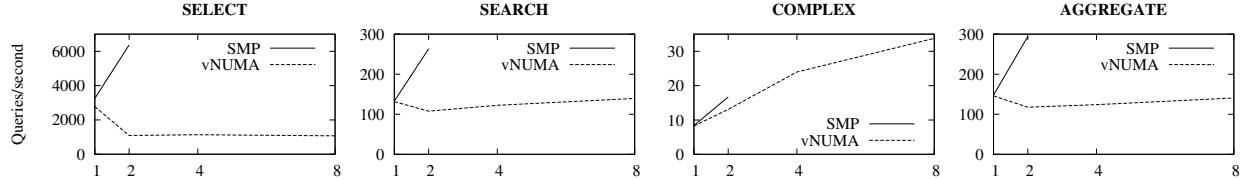


Figure 12: Database benchmark performance summary. Horizontal axes represent number of nodes.

and synchronisation latencies. If one considers that each query involves at least a certain number of these high-latency events, then the maximum query throughput per node is inversely proportional to the number and cost of those events.

A breakdown of processor time usage for **SELECT** shows that only 14 % of available processor time is used for user-level computation, which explains why the four nodes cannot match the performance of a single node. Another 12 % is spent idle, which occurs when the PostgreSQL server processes are waiting to acquire locks. DSM stalls account for 57 % of processor time, with three-quarters of that being in userspace and specifically in the PostgreSQL server processes, and the other quarter in the Linux kernel. There is 9 % overhead for logging writes for the write-update protocol, and 2 % virtualization overhead (while **SELECT** normally experiences high virtualization overheads, the fact that it is only running 14 % of the time makes the virtualization overhead insignificant).

Further analysis, using performance counters, confirms that the major overheads are related to locking within PostgreSQL. The system uses multiple layers of locks: spinlocks, “lightweight” locks built on spinlocks, and heavyweight locks built on lightweight locks. Importantly, each heavyweight lock does not use its own lightweight lock, but there are a small number of contiguous lightweight locks which are used for protecting data about all of the heavyweight locks in the system. Thus, contention for this small number of lightweight locks can hamper the scalability of all heavyweight locks. In addition to this bottleneck, the multi-layer design substantially increases the potential overheads when lock contention occurs.

While this result is disappointing for vNUMA, it is not reasonable to extrapolate from PostgreSQL and assume that all database software will experience such severe locking problems. Since vNUMA can provide high levels of read replication and caching — and potentially a large amount of distributed RAM that may be faster than disk — designs that allow lock-free read accesses to data, such as via read-copy-update techniques [12,25], could theoretically provide very good performance. In this case, kernel performance would again become the ultimate challenge.

6 Related Work

Ivy [23] is the ancestor of most modern DSM systems. Ivy introduced the basic write-invalidate DSM protocol that forms an integral part of vNUMA’s protocol. Mirage [11] moved the DSM system into the OS kernel, thus improving transparency. It also attempted to address the page thrashing problem, which was mentioned earlier in Section 4.1. Ivy and Mirage were followed by a large number of similar systems [28].

Munin [5] was the first system to leverage release consistency to allow multiple simultaneous writers. Aside from release consistency, other systems have also implemented entry consistency (Midway [4]), scope consistency (JIAJIA [9], Brazos [33]) and view-based consistency (VODCA [15]), which further relax the consistency model by associating specific objects with critical sections. However, all of these systems rely on the programmer to adhere to a particular memory synchronisation model, and thus they are not suitable for transparent execution of unmodified applications.

Recently there has also been much interest in virtualization, with systems such as Xen, VMware ESX Server and Microsoft Virtual Server making inroads in the enterprise. The majority of hypervisors are designed for the purposes of server consolidation, allowing multiple OS instances to be co-located on a single physical computer. vNUMA is, in a sense, the opposite, allowing multiple physical computers to host a single OS instance.

Since our initial work [7], three other systems have emerged which apply similar ideas to vNUMA: Virtual Iron’s VFe hypervisor [34], ScaleMP’s vSMP [32] and the University of Tokyo’s Virtual Multiprocessor [18]. While these systems all combine virtualization with distributed shared memory, they are limited in scope and performance, and do not address many of the challenges that this work addresses. In particular, both VFe and the Tokyo system use simpler virtualization schemes and distributed shared memory protocols, resulting in severe performance limitations, especially in the case of Virtual Multiprocessor. Virtual Iron attempted to address some of these performance issues by using high-end hardware, such as InfiniBand rather than Gigabit Ethernet. However, this greatly increases the cost of such a system, and limits the target market. Virtual Iron has since aban-

doned the product for commercial reasons, which largely seems to stem from its dependence on such high-end hardware. vNUMA, in contrast, demonstrates how novel techniques can achieve good performance on commodity hardware.

Little is known about vSMP, other than that it runs on x86-64 hardware and also relies on InfiniBand. The company claims scalability to 128 nodes, but only publishes benchmarks showing the performance of (single-threaded) SPEC benchmarks. No real comparison with vNUMA is possible with the information available.

7 Conclusions and Future Work

We have presented vNUMA, a system that uses virtualization to present a small cluster as a shared-memory multiprocessor, able to support legacy SMP/NUMA operating-system and multiprocessor applications. This approach provides a higher level of transparency than classical software DSM systems. Implementation in the hypervisor also has the advantage that many operations can be implemented more efficiently, and can make use of all the features of the underlying processor architecture. However, a faithful mirroring of the underlying ISA is required.

The different trade-offs resulted in protocols and implementation choices that are quite different from most existing DSM systems. Specifically, we developed a protocol utilizing broadcast of write-updates, which adaptively transitions between write-update/multiple-writer, write-update/single-writer and write-invalidate modes of operation. We also designed a deterministic incremental merge scheme that can provide true write coherence.

The evaluation showed that vNUMA scales significantly better than TreadMarks on HPC workloads, and equal to `distcc` on compiles. Database benchmarks showed the limitations of vNUMA for workloads which make extensive use of locks.

At the time this project was commenced (2002), Itanium was envisaged as the commodity system of the future, a 64-bit replacement of x86. This clearly has not happened, and as such, hardware supporting the present vNUMA implementation is not exactly considered “commodity”, widespread deployment of Itanium systems in HPC environments notwithstanding. We are therefore investigating a port of vNUMA to AMD64 platforms. Some optimisations, such as those described in Section 4.3, will not apply there, but there is scope for other architecture-specific optimisations.

References

- [1] James K. Archibald. A cache coherence approach for large multiprocessor systems. In *2nd Int. Conf. Supercomp.*, pages 337–345, 1988.
- [2] Amnon Barak, Oren La’adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for Linux. In *Proceedings of Linux Expo ’99*, pages 95–100, 1999.
- [3] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *PPOPP*, pages 168–176. ACM, 1990.
- [4] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, 1991.
- [5] John B. Carter. Design of the Munin distributed shared memory system. *J. Parall. & Distr. Comput.*, 29:219–227, 1995.
- [6] Matthew Chapman. *vNUMA: Virtual Shared-Memory Multiprocessors*. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Mar 2009.
- [7] Matthew Chapman and Gernot Heiser. Implementing transparent shared memory on clusters using virtual machines. In *2005 USENIX*, pages 383–386, Anaheim, CA, USA, Apr 2005.
- [8] Xavier Defago, Andre Schiper, and Peter Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *Comput. Surveys*, 36:372–421, 2004.
- [9] M. Rasit Eskicioglu, T. Anthony Marsland, Weiwu Hu, and Weisong Shi. Evaluation of JIAJIA software DSM system on high performance computer architectures. In *32nd HICSS*, 1999.
- [10] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Mar 1994.
- [11] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. In *12th SOSP*, pages 211–223, 1989.
- [12] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *3rd OSDI*, pages 87–100, New Orleans, LA, USA, Feb 1999.
- [13] Ganesh Gopalakrishnan, Dilip Khandekar, Ravi Kuramkote, and Ratan Nalumasu. Case studies in symbolic model checking. Technical Report UUCS-94-009, Dept of Computer Science, University of Utah, 1994.
- [14] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — a system implementor’s tale. In *2005*

- USENIX, pages 264–278, Anaheim, CA, USA, Apr 2005.
- [15] Zhiyi Huang, Wenguang Chen, Martin Purvis, and Weimin Zheng. VODCA: View-oriented, distributed, cluster-based approach to parallel computing. In *6th CCGrid*, 2001.
- [16] Intel Corp. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, Oct 2002. <http://www.intel.com/design/itanium2/documentation.htm>.
- [17] Intel Corp. *Itanium Architecture Software Developer's Manual*, Jan 2006. <http://www.intel.com/design/itanium2/documentation.htm>.
- [18] Kenji Kaneda. Virtual machine monitor for providing a single system image. <http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/dvm/>.
- [19] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *1994 Winter USENIX*, pages 115–131, 1994.
- [20] R.E. Kessler and Miron Livny. An analysis of distributed shared memory algorithms. In *9th ICDCS*, pages 498–505, 1989.
- [21] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21:558–565, 1978.
- [22] Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: soft layering for virtual machines. In Y-C Chung and J Morris, editors, *13th IEEE Asia-Pacific Comp. Syst. Arch. Conf*, pages 1–9, Hsinchu, Taiwan, Aug 2008. IEEE Computer Society Press.
- [23] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *Trans. Comp. Syst.*, 7:321–59, 1989.
- [24] Daniel J. Magenheimer and Thomas W. Christian. vBlades: Optimised paravirtualisation for the Itanium processor family. In *3rd USENIX-VM*, pages 73–82, 2004.
- [25] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *10th IASTED Int. Conf. Parallel. & Distr. Comput. & Syst.*, Las Vegas, NV, USA, Oct 1998.
- [26] Message Passing Interface Forum. MPI: A message-passing interface standard, Nov 2003.
- [27] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, David Margery, Jean-Yves Berthou, and Isaac D. Scherson. Kerrighed and data parallelism: cluster computing on single system image operating systems. In *6th Int. Conf. Cluster Comput.*, pages 277–286, 2004.
- [28] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Comp.*, 24(8):52–60, Aug 1991.
- [29] Martin Pool. distcc, a fast free distributed compiler. In *5th Linux Conf.Au*, Jan 2004. <http://distcc.samba.org/>.
- [30] PostgreSQL Global Development Group. PostgreSQL database software. <http://www.postgresql.org/>.
- [31] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *11th ISCA*, pages 340–347, 1984.
- [32] The Versatile SMP (vSMP) architecture and solutions based on vSMP Foundation. ScaleMP White Paper.
- [33] Evan Speight and John K. Bennett. Brazos: A third generation DSM system. In *1st USENIX Windows NT WS*, pages 95–106, 1997.
- [34] Alex Vasilevsky. Linux virtualization on Virtual Iron VFe. In *2005 Ottawa Linux Symp.*, Jul 2005.
- [35] Bruce J. Walker. Open single system image (openSSI) Linux cluster project. <http://www.openssi.org/ssi-intro.pdf>, accessed on 30th September 2008.
- [36] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd ISCA*, pages 24–36, 1995.
- [37] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software write detection for a distributed shared memory. In *1st OSDI*, pages 87–100, 1994.

ShadowNet: A Platform for Rapid and Safe Network Evolution

Xu Chen Z. Morley Mao
University of Michigan

Jacobus Van der Merwe
AT&T Labs - Research

Abstract

The ability to rapidly deploy new network services, service features and operational tools, without impacting existing services, is a significant challenge for all service providers. In this paper we address this problem by the introduction of a platform called *ShadowNet*. ShadowNet exploits the strong separation provided by modern computing and network equipment between logical functionality and physical infrastructure. It allows logical topologies of computing servers, network equipment and links to be dynamically created, and then instantiated to and managed on the physical infrastructure. ShadowNet is a sharable, programmable and composable infrastructure, consisting of carrier-grade equipment. Furthermore, it is a fully operational network that is connected to, but functionally separate from the provider production network. By exploiting the strong separation support, ShadowNet allows multiple technology and service trials to be executed in parallel in a realistic operational setting, without impacting the production network. In this paper, we describe the ShadowNet architecture and the control framework designed for its operation and illustrate the utility of the platform. We present our prototype implementation and demonstrate the effectiveness of the platform through extensive evaluation.

1 Introduction

Effecting network change is fundamentally difficult. This is primarily due to the fact that modern networks are inherently shared and multi-service in nature, and any change to the network has the potential to negatively impact existing users and services. Historically, production quality network equipment has also been proprietary and closed in nature, thus further raising the bar to the introduction of any new network functionality. The negative impact of this state of affairs has been widely recognized as impeding innovation and evolution [23]. Indeed at a macro-level, the status quo has led to calls for a clean slate redesign of the Internet which in turn has produced efforts such as GENI [3] and FEDERICA [2].

In the work presented in this paper we recognize that at a more modest micro-level, the same fundamental problem, *i.e.*, the fact that network change is inherently difficult, is a major *operational* concern for service providers. Specifically, the introduction of new services

or service features typically involves long deployment cycles: configuration changes to network equipment are meticulously lab-tested before staged deployments are performed in an attempt to reduce the potential of any negative impact on existing services. The same applies to any new tools to be deployed in support of network management and operations. This is especially true as network management tools are evolving to be more sophisticated and capable of controlling network functions in an automated closed-loop fashion [25, 9, 7]. The operation of such tools depends on the actual state of the network, presenting further difficulties for testing in a lab environment due to the challenge of artificially recreating realistic network conditions in a lab setting.

In this paper we address these concerns through a platform called *ShadowNet*. ShadowNet is designed to be an operational trial/test network consisting of ShadowNet *nodes* distributed throughout the backbone of a tier-1 provider in the continental US. Each ShadowNet node is composed of a collection of carrier-grade equipment, namely routers, switches and servers. Each node is connected to the Internet as well as to other ShadowNet nodes via a (virtual) backbone.

ShadowNet provides a *sharable, programmable and composable* infrastructure to enable the rapid trial or deployment of new network services or service features, or evaluation of new network management tools in a realistic operational network environment. Specifically, via the Internet connectivity of each ShadowNet node, traffic from arbitrary end-points can reach ShadowNet. ShadowNet connects to and interacts with the provider backbone much like a customer network would. As such the “regular” provider backbone, just like it would protect itself from any other customers, is isolated from the testing and experimentation that take place within ShadowNet. In the first instance, ShadowNet provides the means for testing services and procedures for subsequent deployment in a (separate) production network. However, in time we anticipate ShadowNet-like functionality to be provided by the production network itself to directly enable rapid but safe service deployment.

ShadowNet has much in common with other test networks [10, 27, 22]: (i) ShadowNet utilizes virtualization and/or partitioning capabilities of equipment to enable *sharing* of the platform between different concurrently running trials/experiments; (ii) equipment in ShadowNet

nodes are *programmable* to enable experimentation and the introduction of new functionality; (iii) ShadowNet allows the dynamic *composition* of test/trial topologies.

What makes ShadowNet unique, however, is that this functionality is provided in an *operational network* on *carrier-grade equipment*. This is critically important for our objective to provide a rapid service deployment/evaluation platform, as technology or service trials performed in ShadowNet should mimic technology used in the provider network as closely as possible. This is made possible by recent vendor capabilities that allow the partitioning of physical routers into subsets of resources that essentially provide logically separate (smaller) versions of the physical router [16].

In this paper, we describe the ShadowNet architecture and specifically the ShadowNet control framework. A distinctive aspect of the control framework is that it provides a clean separation between the *physical-level* equipment in the testbed and the *user-level* slice specifications that can be constructed “within” this physical platform. A *slice*, which encapsulates a service trial, is essentially a container of the service design including device connectivity and placement specification. Once instantiated, a slice also contains the allocated physical resources to the service trial. Despite this clean separation, the partitioning capabilities of the underlying hardware allows virtualized equipment to be largely indistinguishable from their physical counterparts, except that they contain fewer resources. The ShadowNet control framework provides a set of interfaces allowing users to programmatically interact with the platform to manage and manipulate their slices.

We make the following contributions in this work:

- Present a sharable, programmable, and composable network architecture which employs strong separation between user-level topologies/slices and their physical realization (§2).
- Present a network control framework that allows users to manipulate their slices and/or the physical resource contained therein with a simple interface (§3).
- Describe physical-level realizations of user-level slice specifications using carrier-grade equipment and network services/capabilities (§4).
- Present a prototype implementation (§5) and evaluation of our architecture (§6).

2 ShadowNet overview

In this paper, we present ShadowNet which serves as a platform for rapid and safe network change. The primary goal of ShadowNet is to allow the rapid composition of distributed computing and networking resources, contained in a slice, realized in carrier-grade facilities which can be utilized to introduce and/or test new ser-

vices or network management tools. The ShadowNet control framework allows the network-wide resources that make up each slice to be managed either collectively or individually.

In the first instance, ShadowNet will limit new services to the set of resources allocated for that purpose, *i.e.*, contained in a slice. This would be a sufficient solution for testing and trying out new services in a realistic environment before introducing such services into a production network. Indeed our current deployment plans espouse this approach with ShadowNet as a separate overlay facility [24] connected to a tier-1 production network. Longer term, however, we expect the base functionality provided by ShadowNet to evolve into the production network and to allow resources and functionality from different slices to be gracefully merged under the control of the ShadowNet control framework.

In the remainder of this section we first elaborate on the challenges network service providers face in effecting network changes. We describe the ShadowNet architecture and show how it can be used to realize a sophisticated service. Several experimental network platforms are compared against it, and we show that ShadowNet is unique in terms of its ability to provide realistic network testing. Finally we describe the architecture of the primary system component, namely the ShadowNet controller.

2.1 Dealing with network change

There are primarily three drivers for changes in modern service provider networks:

Growth demands: Fueled by an increase in broadband subscribers and media rich content, traffic volumes on the Internet continue to show double digit growth rates year after year. The implication of this is that service providers are required to increase link and/or equipment capacities on a regular basis, even if the network functionality essentially stays the same.

New services and technologies: Satisfying customer needs through new service offerings is essential to the survival of any network provider. “Service” here spans the range from application-level services like VoIP and IPTV, connectivity services like VPNs and IPv4/IPv6 transport, traffic management services like DDoS mitigation or content distribution networks (CDNs), or more mundane (but equally important and complicated) service features like the ability to signal routing preferences to the provider or load balancing features.

New operational tools and procedures: Increasing use of IP networks for business critical applications is leading to growing demands on operational procedures. For example, end-user applications are often very intolerant of even the smallest network disruption, leading to the

deployment of methods to decrease routing convergence in the event of network failures. Similarly, availability expectations, in turn driven by higher level business needs, make regularly planned maintenance events problematic, leading to the development of sophisticated operational methods to limit their impact.

As we have alluded to already, the main concern of any network change is that it might have an impact on existing network services, because networks are inherently shared with known and potentially unknown dependencies between components. An example would be the multi-protocol extensions to BGP to enable MPLS-VPNs or indeed any new protocol family. The change associated with rolling out a new extended BGP stack clearly has the potential to impact existing IPv4 BGP interactions, as bugs in new BGP software could negatively impact the BGP stack as a whole.

Note also that network services and service features are normally “cumulative” in the sense that once deployed and used, network services are very rarely “switched off”. This means that over time the dependencies and the potential for negative impact only increases rather than diminishes.

A related complication associated with any network change, especially for new services and service features, is the requirement for corresponding changes to a variety of operational support systems including: (i) configuration management systems (new services need to be configured typically across many network elements), (ii) network management systems (network elements and protocols need to be monitored and maintained), (iii) service monitoring systems (for example to ensure that network-wide service level agreements, *e.g.*, loss, delay or video quality, are met), (iv) provisioning systems (*e.g.*, to ensure the timely build-out of popular services). ShadowNet does not address these concerns per se. However, as described above, new operational solutions are increasingly more sophisticated and automated, and ShadowNet provides the means for safely testing out such functionality in a realistic environment.

Our ultimate goal with the ShadowNet work is to develop mechanisms and network management primitives that would allow new services and operational tools to be safely deployed directly in production networks. However, as we describe next, in the work presented here we take the more modest first step of allowing such actions to be performed in an operational network that is separate from the production network, which is an important transitional step.

2.2 ShadowNet architecture

Different viewpoints of the ShadowNet network architecture are shown in Figures 1(a) and (b). Figure 1(a) shows the topology from the viewpoint of the tier-1

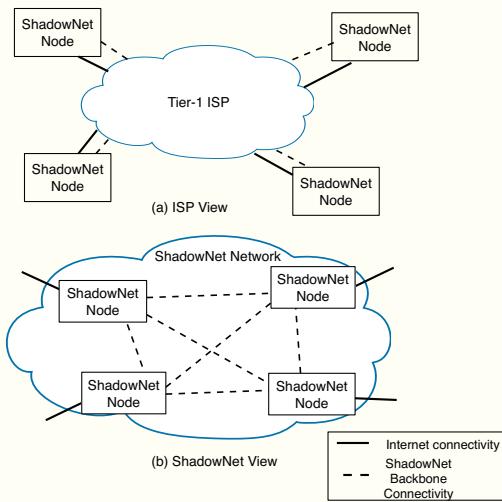


Figure 1: ShadowNet network viewpoints

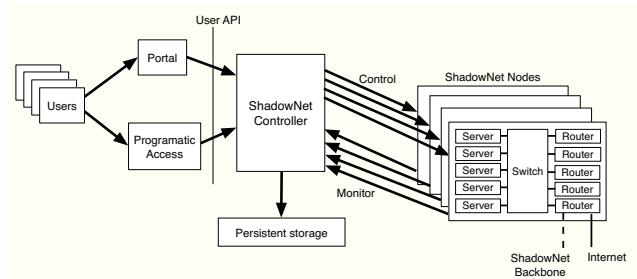


Figure 2: ShadowNet functional architecture

provider. ShadowNet nodes connect to the provider network, but are essentially separate from it. Each ShadowNet node has connectivity to other ShadowNet nodes as well as connectivity to the Internet. As shown in Figure 1(b), connectivity to other ShadowNet nodes effectively creates an overlay network [24] to form a virtual backbone among the nodes. Via the provided Internet connectivity, the ShadowNet address space is advertised (*e.g.*, using BGP) first to the provider network and then to the rest of the Internet. Thus ShadowNet effectively becomes a small provider network itself, *i.e.*, a *shadow* of the provider network.

The ShadowNet functional architecture is shown in Figure 2. Each ShadowNet node contains different types of computing and networking devices, such as servers, routers, and switches. Combined with the network connectivity received from the ISP, they complete the physical resource for ShadowNet. ShadowNet manages the physical resources and enables its users to share them. The devices provide virtualization/partitioning capabilities so that multiple logical devices can share the same underlying physical resource. For example, modern routers allow router resources to be partitioned so that several logical routers can be configured to run simultaneously and separately on a single physical router [16].

(Note that modern routers are also programmable in both control and data planes [18].) Logical interfaces can be multiplexed from one physical interface via configuration and then assigned to different logical routers. We also take advantage of virtual machine technology to manage server resources [5]. This technology enables multiple operating systems to run simultaneously on the same physical machine and is already heavily used in cloud computing and data-center environments. To facilitate sharing connectivity, the physical devices in each ShadowNet node are connected via a configurable switching layer, which shares the local connectivity, for example using VLANs. The carrier-supporting-carrier capabilities enabled by MPLS virtual private networks (VPNs) [11, 15] offer strong isolation and are therefore an ideal choice to create the ShadowNet backbone.

As depicted in Figure 2, central to ShadowNet functionality is the *ShadowNet Controller*. The controller facilitates the specification and instantiation of a service trial in the form of a slice owned by a user. It provides a programmatic application programming interface (API) to ShadowNet users, allowing them to create the topological setup of the intended service trial or deployment. Alternatively users can access ShadowNet through a Web-based portal, which in turn will interact with the ShadowNet Controller via the user-level API. The ShadowNet Controller keeps track of the physical devices that make up each ShadowNet node by constantly monitoring them, and further manages and manipulates those physical devices to realize the user-level APIs, while maintaining a clean separation between the abstracted slice specifications and the way they are realized on the physical equipment. The user-level APIs also enable users to dynamically interact with and manage the physical instantiation of their slices. Specifically, users can directly access and configure each instantiated logical device.

ShadowNet allows a user to deactivate individual devices in a slice or the slice as a whole, by releasing the allocated physical resources. ShadowNet decouples the persistent state from the instantiated physical devices, so that the state change associated with a device in the specification is maintained even if the physical instantiation is released. Subsequently, that device in the specification can be re-instantiated (assuming that sufficient resources are available), the saved state restored and thus the user perceived slice remains intact. For example, the configuration change made by the user to a logical router can be maintained and applied to a new instantiated logical router, even if the physical placement of that logical device is different.

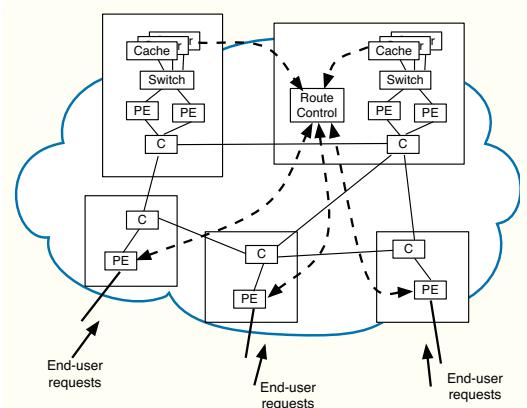


Figure 3: Usage scenario: load-aware anycast CDN.

2.3 Using ShadowNet

In this section we briefly describe an example usage scenario that illustrates the type of sophisticated network services that can be tested using the ShadowNet infrastructure. We discuss the requirements for testing these services and explain why existing platforms fall short in these scenarios.

Assume that ShadowNet is to be used to run a customer trial of a *load-aware anycast content distribution network (CDN)* [9]. Figure 3 depicts how all the components of such a CDN can be realized on the ShadowNet platform. Specifically, a network, complete with provider edge (PE) and core (C) routers, can be dynamically instantiated to represent a small backbone network. Further, servers in a subset of the ShadowNet nodes can be allocated and configured to serve as content caches. A load-aware anycast CDN utilizes route control to inform BGP selection based on the cache load, *i.e.*, using BGP, traffic can be steered away from overloaded cache servers. In ShadowNet, this BGP speaking route control entity can be instantiated on either a server or a router depending on the implementation. Appropriate configuration/implementation of BGP, flow-sampling, and server load monitoring complete the infrastructure picture. Finally, actual end-user requests can be directed to this infrastructure, *e.g.*, by resolving a content URL to the anycast address(es) associated with and advertised by the CDN contained in the ShadowNet infrastructure.

Using this example we can identify several capabilities required of the ShadowNet infrastructure to enable such realistic service evaluation (see Table 1): (i) to gain confidence in the equipment used in the trial it should be the same as, or similar to, equipment used in the production network (*production-grade devices*); (ii) to thoroughly test load feedback mechanisms and traffic steering algorithms, it requires participation of significant numbers of customers (*realistic workloads*); (iii) this in turn requires sufficient network capacity (*high capacity backbone*); (iv) realistic network and CDN functionality

	SN	EL	PL	VNI
Production grade devices	Y	N	N	N
Realistic workloads	Y	N	Y	Y
High capacity backbone	Y	N	N	Y
Geographical coverage	Y	N	Y	Y
Dynamic reconfiguration	Y	N	N	N

Table 1: Capability comparison between ShadowNet (SN), EmuLab (EL), PlanetLab (PL) and VNI (VN)

require realistic network latencies and geographic distribution (*geographic coverage*); (v) finally, the CDN control framework could dynamically adjust the resources allocated to it based on the offered load (*dynamic reconfiguration*).

While ShadowNet is designed to satisfy these requirements, other testing platforms, with different design goals and typical usage scenarios, fall short in providing such support, as we describe next.

Emulab achieves flexible network topology through emulation within a central testbed environment. There is a significant gap between emulation environments and real production networks. For example, software routers typically do not provide the same throughput as production routers with hardware support. As Emulab is a closed environment, it is incapable of combining real Internet workload into experiments. Compared to Emulab, the ShadowNet infrastructure is distributed, thus the resource placement in ShadowNet more closely resembles future deployment phases. In Emulab, an experiment in a slice is allocated a fixed set of resources during its life cycle — a change of specification would require a “reboot” of the slice. ShadowNet, on the other hand, can change the specification *dynamically*. In the CDN example, machines for content caches and network links can be dynamically spawned or removed in response to increased or decreased client requests.

PlanetLab has been extremely successful in academic research, especially in distributed monitoring and P2P research. It achieves its goal of amazing geographical coverage, spanning nodes to all over the globe, obtaining great end-host visibility. The PlanetLab nodes, however, are mostly connected to educational networks without abundant upstream or downstream bandwidth. PlanetLab therefore lacks the capacity to realize a capable *backbone* between PlanetLab nodes. ShadowNet, on the other hand, is built upon a production ISP network, having its own virtual backbone with bandwidth and latency guarantees. This pushes the tested service closer to the core of the ISP network, where the actual production service would be deployed.

VNI is closely tied with PlanetLab, but utilizes Internet2 to provide a realistic backbone. Like Emulab

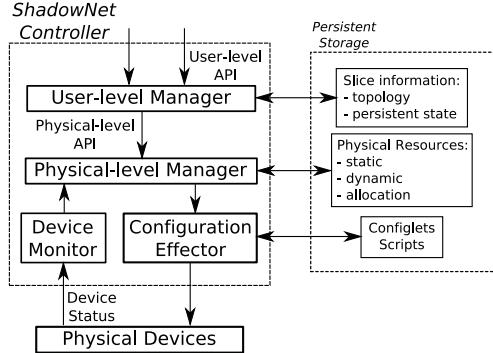


Figure 4: The ShadowNet controller

and PlanetLab, VNI runs software routers (XORP and Click), the forwarding capacity of which lags behind production devices. This is mostly because its focus is to use commodity hardware to evaluate new Internet architectures, which is different from the service deployment focus of ShadowNet. VNI and PlanetLab are based on the same control framework. Similar to Emulab, it lacks the capability of changing slice configurations dynamically, *i.e.*, not closing the loop for more adaptive resource management, a functionality readily available in ShadowNet.

2.4 The ShadowNet Controller

The ShadowNet controller consists of a user-level manager, a physical-level manager, a configuration effector and a device monitor, as shown in Figure 4. We describe each component below. The current ShadowNet design utilizes a centralized controller that interacts with and controls all ShadowNet nodes.

2.4.1 User-level manager

The user-level manager is designed to take the input of user-level API calls. Each API call corresponds to an action that the users of ShadowNet are allowed to perform. A user can create a topological specification of a service trial (§3.1), instantiate the specification to physical resources (§3.2), interact with the allocated physical resources (§3.3), and deactivate the slice when the test finishes (§3.4). The topology specification of a slice is stored by the user-level manager in persistent storage, so that it can be retrieved, revived and modified over time. The user-level manager also helps maintain and manage the saved persistent state from physical instantiations (§3.3). By retrieving saved states and applying them to physical instantiations, advanced features, like device duplication, can be enabled (§3.5).

The user-level manager is essentially a network service used to manipulate configurations of user experiments. We allow the user-level manager to be accessed from within the experiment, facilitating network control

in a closed-loop fashion. In the example shown in Figure 3, the route control component in the experiment can dynamically add content caches when user demand is high by calling the user-level API to add more computing and networking resource via the user-level manager.

2.4.2 Physical-level manager

The physical-level manager fulfills requests from the user-level manager in the form of physical-level API calls by manipulating the physical resources in ShadowNet. To do this, it maintains three types of information: 1) “static” information, such as the devices in each ShadowNet node and their capabilities; 2) “dynamic” information, e.g., the online status of all devices and whether any interface modules are not functioning; 3) “allocation” information, which is the up-to-date usage of the physical resources. Static information is changed when new devices are added or old devices are removed. Dynamic information is constantly updated by the device monitor. The three main functions of the physical-level manager is to configure physical devices to spawn virtualized *device slivers* (§4.1) for the instantiation of user-level devices (§4.1.1) and user-level connectivities (§4.1.2), to manage their states (§4.4) and to delete existing instantiated slivers. A *sliver* is a share of the physical resource, e.g., a virtual machine or a sliced physical link. The physical-level manager handles requests, such as creating a VM, by figuring out the physical device to configure and how to configure it. The actual management actions are performed via the configuration effector module, which we describe next.

2.4.3 Configuration effector

The configuration effector specializes in realizing configuration changes to physical devices. *Configlets* are parametrized configuration or script templates, saved in the persistent storage and retrieved on demand. To realize the physical-level API calls, the physical-level manager decides the appropriate configlet to use and generates parameters based on the request and the physical resource information. The configuration effector executes the configuration change on target physical devices.

2.4.4 Device monitor

A device monitor actively or passively determines the status of physical devices or components and propagates this “dynamic” information to the physical-level manager. Effectively, the device monitor detects any physical device failures in real time. As the physical-level manager receives the update, it can perform appropriate actions to mitigate the failure. The goal is to minimize any inconsistency of physical instantiation and user specifications. We detail the techniques in §4.5. Device or component recovery can be detected as well, and as

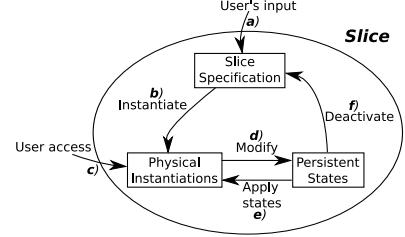
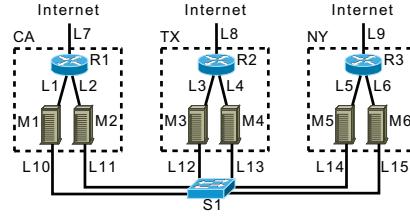


Figure 5: The slice life cycle



```

$SL = AddUsrSlice();
$S1 = AddUsrSwitch($SL);
$R1 = AddUsrRouter($SL, "CA");
$M1 = AddUsrMachine($SL, "CA", "Debian");
$M2 = AddUsrMachine($SL, "CA", "Windows");
$L1 = AddUsrLink($M1,$R1); # similar for M2
$L10 = AddUsrLink($M1,$S1); # similar for M2
$L7 = AddToInternet($R1, "141.212.111.0/24");
# similar for "TX" and "NY"
  
```

Figure 6: Example of user-level API calls

such the recovered resource can again be considered usable by the physical-level manager.

3 Network service in a slice

A user of ShadowNet creates a service topology in the form of a slice, which is manipulated through the user-level API calls supported by the ShadowNet controller. The three layers embedded in a slice and the interactions among them are depicted in Figure 5 and detailed below. In this section, we outline the main user-exposed functionalities that the APIs implement.

3.1 Creating user-level specification

To create a new service trial, an authorized user of ShadowNet can create a slice. As a basic support, and usually the first step to create the service, the user specifies the topological setup through the user-level API (*a* in Figure 5). As an example, Figure 6 depicts the intended topology of a hypothetical slice and the API call sequence that creates it.

The slice created acts like a placeholder for a collection of *user-level objects*, including devices and connectivities. We support three generic types of user-level devices (UsrDevice): router (UsrRouter), machine (UsrMachine), and switch (UsrSwitch). Two UsrDevices can be connected to each other via a user-level link (UsrLink). User-level interfaces (UsrInt) can be added to

a UsrDevice explicitly by the slice owner; however, in most cases, they are created implicitly when a UsrLink is added to connect two UsrDevices.

Functionally speaking, a UsrMachine (*e.g.*, M_1 in Figure 6) represents a generic computing resource, where the user can run service applications. A UsrRouter (*e.g.*, R_1) can run routing protocols, forward and filter packets, *etc.* Further, UsrRouters are programmable, allowing for custom router functionality. A UsrLink (*e.g.*, L_1) ensures that when the UsrDevice on one end sends a packet, the UsrDevice on the other end will receive it. A UsrSwitch (*e.g.*, S_1) provides a single broadcast domain to the UsrDevices connecting to it. ShadowNet provides the capability and flexibility of putting geographically dispersed devices on the same broadcast domain. For example, M_1 to M_6 , although specified in different locations, are all connected to UsrSwitch S_1 . Besides internal connectivity among UsrDevices, ShadowNet can drive live Internet traffic to a service trial by allocating a public IP prefix for a UsrInt on a UsrDevice. For example, L_7 is used to connect R_1 to the Internet, allocating an IP prefix of 141.212.111.0/24.

Besides creating devices and links, a user of ShadowNet can also associate properties with different objects, *e.g.*, the OS image of a UsrMachine and the IP addresses of the two interfaces on each side of a UsrLink. As a distributed infrastructure, ShadowNet allows users to specify location preference for each device as well, *e.g.*, California for M_1 , M_2 and R_1 . This location information is used by the physical layer manager when instantiation is performed.

3.2 Instantiation

A user can instantiate some or all objects in her slice onto physical resources (b in Figure 5). From this point on, the slice not only contains abstracted specification, but also has associated physical resources that the instantiated objects in the specification are mapped to.

ShadowNet provides two types of instantiation strategies. First, a user can design a full specification for the slice and instantiate all the objects in the specification together. This is similar to what Emulab and VINI provide. As a second option, user-level objects in the specification can be instantiated upon request at any time. For example, they can be instantiated on-the-fly as they are added to the service specification. This is useful for users who would like to build a slice interactively and/or modify it over time, *e.g.*, extend the slice resources based on increased demand.

Unlike other platforms, such as PlanetLab and Emulab, which intend to run as many “slices” as possible, ShadowNet limits the number of shares (slivers) a physical resource provides. This simplifies the resource al-

location problem to a straightforward availability check. We leave more advanced resource allocation methods as future work.

3.3 Device access & persistent slice state

ShadowNet allows a user to access the physical instantiation of the UsrDevices and UsrLinks in her slice, *e.g.*, logging into a router or tapping into a link (c in Figure 5). This support is necessary for many reasons. First, the user needs to install software on UsrMachines or UsrRouters and/or configure UsrRouters for forwarding and filtering packets. Second, purely from an operational point of view, operators usually desire direct access to the devices (*e.g.*, a terminal window on a server, or command line access to a router).

For UsrMachines and UsrRouters, we allow users to log into the device and make any changes they want (§4.3). For UsrLinks and UsrSwitches, we provide packet dump feeds upon request (§4.3). This support is crucial for service testing, debugging and optimization, since it gives the capability and flexibility of sniffing packets at any place within the service deployment without installing additional software on end-points.

Enabling device access also grants users the ability to change the persistent state of the physical instantiations, such as files installed on disks and configuration changes on routers. In ShadowNet, we decouple the persistent states from the physical instantiation. When the physical instantiation is modified, the changed state also become part of the slice (d in Figure 5).

3.4 Deactivation

The instantiated user-level objects in the specification of a slice can be deactivated, releasing the physical instantiations of the objects from the slice by giving them back to the ShadowNet infrastructure. For example, a user can choose to deactivate an under-utilized slice as a whole, so that other users can test their slices when the physical resources are scarce. While releasing the physical resource, we make sure the persistent state is extracted and stored as part of the slice (f in Figure 5). As a result, when the user decides to revive a whole slice or an object in the slice, new physical resources will be acquired and the stored state associated with the object applied to it (e in Figure 5). Operationally speaking, this enables a user to deactivate a slice and reactivate it later, most likely on a different set of resources but still functioning like before.

3.5 Management support

Abstracting the persistent state from the physical instantiation enables other useful primitives in the context of service deployment. If we instantiate a new UsrDevice and apply the state of an existing UsrDevice to it, we ef-

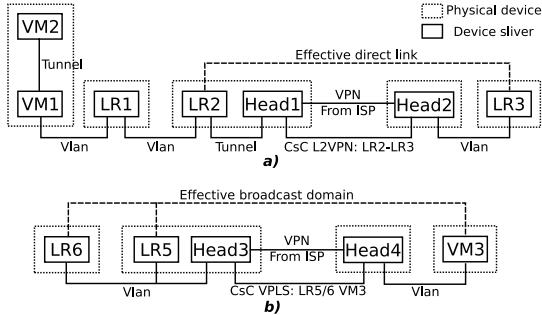


Figure 7: Network connectivity options.

fectively duplicate the existing `UsrDevice`. For example, a user may instantiate a new `UsrMachine` with only the basic OS setup, log into the machine to install necessary application code and configure the OS. With the support provided by `ShadowNet`, she can then spawn several new `UsrMachines` and apply the state of the first machine. This eases the task of creating a cluster of devices serving similar purposes. From the `ShadowNet` control aspect, this separation allows sophisticated techniques to hide physical device failures. For example, a physical router experiences a power failure, while it hosts many logical routers as the instantiation of `UsrRouters`. In this case, we only need to create new instantiations on other available devices of the same type, and then apply the states to them. During the whole process, the slice specification, which is what the user perceives, is intact. Naturally, the slice will experience some downtime as a result of the failure.

4 Physical layer operations

While conceptually similar to several existing systems [10, 27], engineering ShadowNet is challenging due to the strong isolation concept it rests on, the production-grade qualities it provides and the distributed nature of its realization. We describe the key methods used to realize ShadowNet.

4.1 Instantiating slice specifications

The slice specification instantiation is performed by the ShadowNet controller in a fully automated fashion. The methods to instantiate on two types of resource are described as follows.

4.1.1 User-level routers and machines

ShadowNet currently utilizes VirtualBox [5] from Sun Microsystems, and Logical Routers [16] from Juniper Networks to realize UsrMachines and UsrRouters respectively. Each VM and logical router created is considered as a device *sliver*. To instantiate a UsrRouter or a UsrMachine, a ShadowNet node is chosen based on the location property specified. Then all matching physical devices on that node are enumerated for avail-

ability checking, e.g., whether a Juniper router is capable of spawning a new logical router. When there are multiple choices, we distribute the usage across devices in a round-robin fashion. Location preference may be unspecified because the user does not care about where the `UsrDevice` is instantiated, e.g., when testing a router configuration option. In this case, we greedily choose the `ShadowNet` node where that type of device is the least utilized. When no available resource can be allocated, an error is returned.

4.1.2 User-level connectivity

The production network associated with ShadowNet provides both Internet connection and virtual backbone connectivity to each ShadowNet node. We configure a logical router, which we call the *head* router of the ShadowNet node, to terminate these two connections. With the ShadowNet backbone connectivity provided by the ISP, all head routers form a full-mesh, serving as the core routers of ShadowNet. For Internet connectivity, the head router interacts with ISP's border router, *e.g.*, announcing BGP routes.

Connecting device slivers on the same ShadowNet node can be handled by the switching layer of that node. The head routers are used when device slivers across nodes need to be connected. In ShadowNet, we make use of the carrier-supporting-carrier (CsC) capabilities provided by MPLS enabled networks. CsC utilizes the VPN service provided by the ISP, and stacks on top of it another layer of VPN services, running in parallel but isolated from each other. For example, layer-2 VPNs (so called pseudo-wire) and VPLS VPNs can be stacked on top of a layer-3 VPN service [15].

This approach has three key benefits. First, each layer-2 VPN or VPLS instance encapsulates the network traffic within the instance, thus provides strong isolation across links. Second, these are off-the-shelf production-grade services, which are much more efficient than manually configured tunnels. Third, it is more realistic for the users, because there is no additional configuration needed in the logical routers they use. The layer-2 VPN and VPLS options that we heavily use in ShadowNet provides layer-2 connectivity, *i.e.*, with router programmability, any layer-3 protocol besides IP can run on top of it.

Figure 7 contains various examples of enabling connectivity, which we explain in detail next.

UsrLink: To instantiate a UsrLink, the instantiations of the two UsrDevices on the two ends of the UsrLink are first identified. We handle three cases, see Figure 7a). (We consider the UsrLinks connected to a UsrSwitch part of that UsrSwitch, which we describe later):

1) Two slivers are on the same physical device: for example, $VM1$ and $VM2$ are on the same server; $LR2$

and *Head1* are on the same router. In this case, we use local bridging to realize the UsrLink.

2) Two slivers are on the same ShadowNet node, but not the same device: for example, *VM1* and *LR1*, *LR1* and *LR2*. We use a dedicated VLAN on that node for each UsrLink of this type, *e.g.*, *LR1* will be configured with two interfaces, joining two different VLAN segments, one for the link to *VM1*, the other one to *LR2*.

3) Two slivers are on different nodes: for example, *LR2* and *LR3*. In this case, we first connect each sliver to its local head router, using the two methods above. Then the head router creates a layer-2 VPN to bridge the added interfaces, effectively creating a cross-node tunnel connecting the two slivers.

In each scenario above, the types of the physical interfaces that should be used to enable the link are decided, the selected physical interfaces are configured, and the resource usage information of the interfaces is updated.

MPLS-VPN technologies achieve much higher levels of realism over software tunnels, because almost no configuration is required at the end-points that are being connected. For example, to enable the direct link between *LR2* and *LR3*, the layer-2 VPN configuration only happens on *Head1* and *Head2*. As a result, if the user logs into the logical router *LR2* after its creation, she would only see a “physical” interface setup in the configuration, even without IP configured, yet that interface leads to *LR3* according to the layer-2 topology.

User-view switches: Unlike for UsrMachines and UsrRouters, ShadowNet does not allocate user-controllable device slivers for the instantiation of UsrSwitches, but rather provide an Ethernet broadcasting medium. (See Figure 7b).)

To instantiate a UsrSwitch connecting to a set of UsrDevices instantiated on the same ShadowNet node, we allocate a dedicated VLAN-ID on that node and configure those device slivers to join the VLAN (*i.e.*, *LR5* and *LR6*). If the device slivers mapped to the UsrDevices distribute across different ShadowNet nodes, we first recursively bridge the slivers on the same node using VLANs, and then configure one VPLS-VPN instance on each head router (*i.e.*, *Head3* and *Head4*) to bridge all those VLANs. This puts all those device slivers (*i.e.*, *VM3*, *LR5*, *LR6*) onto the same broadcast domain. Similar to layer-2 VPN, this achieves a high degree of realism, for example on *LR5* and *LR6*, the instantiated logical router only shows one “physical” interface in its configuration.

Internet access: We assume that ShadowNet nodes can use a set of prefixes to communicate with any end-points on the Internet. The prefixes can either be announced through BGP sessions configured on the head routers to

the ISP’s border routers, or statically configured on the border routers.

To instantiate a UsrDevice’s Internet connectivity, we first connect the UsrDevice’s instantiation to the head router on the same node. Then we configure the head router so that the allocated prefix is correctly forwarded to the UsrDevice over the established link and the route for the prefix is announced via BGP to the ISP. For example, a user specifies two UsrRouters connecting to the Internet, allocating them with prefix 136.12.0.0/24 and 136.12.1.0/24. The head router should in turn announce an aggregated prefix 136.12.0.0/23 to the ISP border router.

4.2 Achieving isolation and fair sharing

As a shared infrastructure for many users, ShadowNet attempts to minimize the interference among the physical instantiation of different slices. Each virtual machine is allocated with its own memory address space, disk image, and network interfaces. However, some resources, like CPU, are shared among virtual machines, so that one virtual machine could potentially drain most of the CPU cycles. Fortunately, virtual machine technology is developing better control over CPU usage of individual virtual machines [5].

A logical router on a Juniper router has its own configuration file and maintains its own routing table and forwarding table. However, control plane resources, such as CPU and memory are shared among logical routers. We evaluate this impact in §6.3.

The isolation of packets among different UsrLinks is guaranteed by the physical device and routing protocol properties. We leverage router support for packet filtering and shaping, to prevent IP spoofing and bandwidth abusing. The corresponding configuration is made on head routers, where end-users cannot access. For each UsrLink, we impose a default rate-limit (*e.g.*, 10Mbps), which can be upgraded by sending a request via the user-level API. We achieve rate limiting via hardware traffic policers [19] and Linux kernel support [4].

4.3 Enabling device access

Console or remote-desktop access: For each VM running on VirtualBox, a port is specified on the hosting server to enable Remote Desktop protocol for graphical access restricted to that VM. If the user prefers command line access, a serial port console in the VM images is enabled and mapped to a UNIX domain socket on the hosting machine’s file system [5]. On a physical router, each logical router can be configured to be accessible through SSH using a given username and password pair, while confining the access to be within the logical router only.

Though the device slivers of a slice can be connected to the Internet, the management interface of the actual

physical devices in ShadowNet should not be. For example, the IP address of a physical server should be contained within ShadowNet rather than accessible globally. We thus enable users to access the device slivers through one level of indirection via the ShadowNet controller.

Sniffing links: To provide packet traces from a particular UsrLink or UsrSwitch, we dynamically configure a SPAN port on the switching layer of a ShadowNet node so that a dedicated server or a pre-configured VM can sniff the VLAN segment that the UsrLink or UsrSwitch is using. The packet trace can be redirected through the controller to the user in a streaming fashion or saved as a file for future downloading. There are cases where no VLAN is used, *e.g.*, for two logical routers on the same physical router connected via logical tunnel interfaces. In this case, we deactivate the tunnel interfaces and re-instantiate the UsrLink using VLAN setup to support packet capture. This action, however, happens at the physical-level and thus is transparent to the user-level, as the slice specification remains intact.

4.4 Managing state

To extract the state of an instantiated UsrMachine, which essentially is a VM, we keep the hard drive image of the virtual machine. The configuration file of a logical router is considered as the persistent state of the corresponding UsrRouter. Reviving stored state for a UsrMachine can be done by attaching the saved disk image to a newly instantiated VM. On the other hand, UsrRouter state, *i.e.*, router configuration files, need additional processing. For example, a user-level interface may be instantiated as interface `fe-0/1/0.2` and thus appear in the configuration of the instantiated logical router. When the slice is deactivated and instantiated again, the UsrInt may be mapped to a different interface, say `ge-0/2/0.1`. To deal with this complication, we normalize the retrieved configuration and replace physical-dependent information with user-level object handles, and save it as the state.

4.5 Mitigating and creating failures

Unexpected physical device failures can occur, and as an option ShadowNet tries to mitigate failures as quickly as possible to reduce user perceived down time. One benefit of separating the states from the physical instantiation is that we can replace a new physical instantiation with the saved state applied without affecting the user perception. Once a device or a physical component is determined to be offline, ShadowNet controller identifies all instantiated user-level devices associated to it. New instantiations are created on healthy physical devices and saved states are applied if possible. Note that certain users are specifically interested in observing service behavior during failure scenarios. We allow the

users to specify whether they want physical failures to pass through, which is disabling our failure mitigation functionality. On the other hand, failure can be injected by the ShadowNet user-level API, for example tearing down the physical instantiation of a link or a device in the specification to mimic a physical link-down event.

For physical routers, the device monitor performs periodic retrieval of the current configuration files, preserving the states of UsrRouters more proactively. When a whole physical router fails, the controller creates new logical routers with connectivity satisfying the topology on other healthy routers and applies the saved configuration, such as BGP setup. If an interface module fails, the other healthy interfaces on the same router are used instead. Note that the head router is managed in the same way as other logical routers, so that ShadowNet can also recover from router failures where head routers are down.

A physical machine failure is likely more catastrophic, because it is challenging to recover files from a failed machine and it is not feasible to duplicate large files like VM images to the controller. One potential solution is to deploy a distributed file system similar to the Google file system [13] among the physical machines within one ShadowNet node. We leave this type of functionality for future work.

5 Prototype Implementation

In this section, we briefly describe our prototype implementation of the ShadowNet infrastructure, including the hardware setup and management controller.

5.1 Hardware setup

To evaluate our architecture we built two ShadowNet nodes and deployed them locally. (At the time of writing, a four node ShadowNet instance is being deployed as an operational network with nodes in Texas, Illinois, New Jersey and California. Each node has two gigabit links to the production network, one used as regular peering link and the other used as the dedicated backbone.)

Each prototype node has two Juniper M7i routers running JUNOS version 9.0, one Cisco C2960 switch, as well as four HP DL520 servers. The M7i routers are equipped with one or two Gigabit Ethernet PICs (Physical Interface Cards), FastEthernet PIC, and tunneling capability. Each server has two gigabit Ethernet interfaces, and we install VirtualBox in the Linux Debian operating system to host virtual machines. The switch is capable of configuring VLANs and enabling SPAN ports.

In the local deployment, two Cisco 7206 routers act as an ISP backbone. MPLS is enabled on the Cisco routers to provide layer-3 VPN service as the ShadowNet backbone. BGP sessions are established between the head

router of each node and its adjacent Cisco router, enabling external traffic to flow into ShadowNet. We connect the network management interface `fxp0` of Juniper routers and one of the two Ethernet interfaces on machines to a dedicated and separate management switch. These interfaces are configured with private IP addresses, and used for physical device management only, mimicking the out-of-band access which is common in ISP network management.

5.2 Controller

The ShadowNet controller runs on a dedicated machine, sitting on the management switch. The controller is currently implemented in Perl. A Perl module, with all the user-level APIs, can be imported in Perl scripts to create, instantiate and access service specifications, similar to the code shown in Figure 6. A mysql database is running on the same machine as the controller, serving largely, though not entirely, as the persistent storage connecting to the controller. It saves the physical device information, user specifications, and normalized configuration files, etc. We use a different set of tables to maintain physical-level information, *e.g.*, `phy_device_table`, and user-level information, *e.g.*, `usr_link_table`. The Perl module retrieves information from the tables and updates the tables when fulfilling API calls.

The configuration effector of the ShadowNet controller is implemented within the Perl module as well. We make use of the NetConf XML API exposed by Juniper routers to configure and control them. Configlets in the form of parametrized XML files are stored on the controller. The controller retrieves the configuration of the physical router in XML format periodically and when UsrRouters are deactivated. We wrote a specialized XML parser to extract individual logical router configurations and normalize relative fields, such as interface related configurations. The normalized configurations are serialized in text format and stored in the mysql database associating to the specific UsrRouter.

Shell and Perl scripts, which wrap the VirtualBox management interface, are executed on the hosting servers to automatically create VMs, snapshot running VMs, stop or destroy VMs. The configuration effector logs into each hosting server and executes those scripts with the correct parameters. On the servers, we run low-priority cron jobs to maintain a fair amount of default VM images of different OS types. In this case, the request of creating a new VM can be fulfilled fairly quickly, amortizing the overhead across time. We use the following steps to direct the traffic of an interface used by a VM to a particular VLAN. First, we run `tunctl` on the hosting server to create a `tap` interface, which is configured in the VMM to be the “physical” interface of

the VM. Second, we make use of 802.1Q kernel module to create VLAN interfaces on the hosting server, like `eth1.4`, which participates in VLAN4. Finally we use `brctl` to bridge the created tap interface and VLAN interface.

Instead of effecting one configuration change per action, the changes to the physical devices are batched and executed once per device, thus reducing authentication and committing overheads. All devices are manipulated in parallel. We evaluate the effectiveness of these two heuristics in §6.1.

The device monitor module is running as a daemon on the controller machine. SNMP trap messages are enabled on the routers and sent over the management channel to the controller machine. Ping messages are sent periodically to all devices. The two sources of information are processed in the background by the monitoring daemon. When failures are detected, the monitoring module calls the physical-level APIs in the Perl module, which in response populates configlets and executes on the routers to handle failures. An error message is also automatically sent to the administrators.

6 Prototype Evaluation

In this section, we evaluate various aspects of ShadowNet based on two example slices instantiated on our prototype. The user specifications are illustrated on the left side of Figure 8; the physical realization of that specification is on the right. In *Slice1*, two locations are specified, namely LA and NY. On the LA side, one `UsrMachine` (`M1`) and one `UsrRouter` (`R1`) are specified. `R1` is connected to `M1` through a `UsrLink`. `R1` is connected to the Internet through `L2` and to `R2` directly via `L5`. The setup is similar on NY side. We use minimum IP and OSPF configuration to enable the correct forwarding between `M1` and `M2`. *Slice2* has essentially the same setup, except that the two `UsrRouter`s do not have Internet access.

The right side of Figure 8 shows the instantiation of *Slice1* and *Slice2*. `VM1` and `LR1` are the instantiation of `M1` and `R1` respectively. `UsrLink L1` is instantiated as a dedicated channel formed by virtualized interfaces from physical interfaces, `eth1` and `ge-0/1/0`, configured to participate in the same VLAN. To create the `UsrLink L5`, ShadowNet first uses logical tunnel interfaces to connect `LR1` and `LR2` with their head routers, which in turn bridge the logical interfaces using layer-2 VPN.

6.1 Slice creation time

Table 2 shows the creation time for *Slice1*, broken down into instantiation of machine and router, along with database access (DB in the table.) Using a naive approach, the ShadowNet controller needs to spend 82

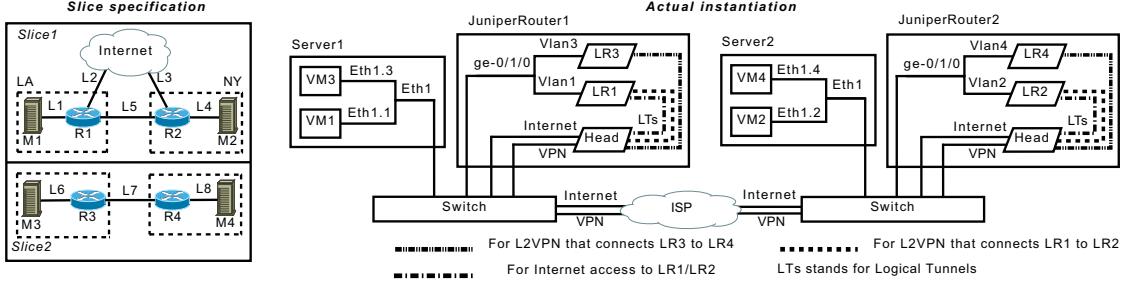


Figure 8: User slices for evaluation

	Router	Machine	DB	Total
Default (ms)	81834	11955	452	94241
Optimized (ms)	6912	5758	452	7364

Table 2: Slice creation time comparison

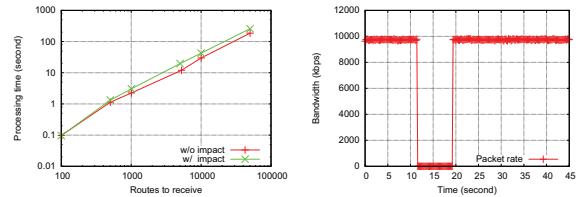
bandwidth (Kbps)	packet size	Observed bandwidth	Delta (%)
56	64	55.9	.18
	1500	55.8	.36
384	64	383.8	.05
	1500	386.0	.52
1544	64	1537.2	.44
	1500	1534.8	.60
5000	1500	4992.2	.16
NoLimit	1500	94791.2	NA

Table 3: Cross-node link stress test

seconds on the physical routers alone by making 13 changes, resulting a 94-second execution time in total. For machine configuration, two scripts are executed for creating the virtual machines, and two for configuring the link connectivity. With the two simple optimization heuristics described in §5.2, the total execution time is reduced to 7.4 seconds. Note that the router and machine configurations are also parallelized, so that we have $\text{total} = \text{DB} + \max(\text{Router}_i, \text{Machine}_j)$. Parallelization ensures that the total time to create a slice does *not* increase linearly with the size of the slice. We estimate creation time for most slices to be within 10 seconds.

6.2 Link stress test

We perform various stress tests to examine ShadowNet’s capability and fidelity. We make L5 the bottleneck link, setting different link constraints using Juniper router’s traffic policer, and then test the observed bandwidth M1 and M2 can achieve on the link by sending packets as fast as possible. Packets are dropped from the head of the queue. The results are shown in Table 3, demonstrating that ShadowNet can closely mimic different link



(a) Impact of shared control (b) Hardware failure recovery planes

Figure 9: Control plane isolation and recovery test. capacities.

When no constraint is placed on L5, the throughput achieved is around 94.8Mbps, shown as “NoLimit” in the table. This is close to maximum, because the routers we used as ISP cores are equipped with FastEthernet interfaces, which have 100Mbps capacity and the VM is specified with 100Mbps virtual interface. Physical gigabit switches are usually not the bottleneck, as we verified that two physical machines on the same physical machines connected via VLAN switch can achieve approximately 1Gbps bandwidth.

As we are evaluating on a local testbed, the jitter and loss rate is almost zero, while the delay is relatively constant. We do not expect this to hold in our wide-area deployment.

6.3 Slice isolation

We describe our results in evaluating the isolation assurance from the perspectives of both the control and data plane.

6.3.1 Control plane

To understand the impact of a stressed control plane on other logical routers, we run software routers, bgpd of zebra, on both M1 and M3. The two software routers are configured to peer with the BGP processes on LR1 and LR3. We load the software routers with BGP routing tables of different sizes, transferred to LR1 and LR3. The BGP event log on the physical router is analyzed by measuring the duration from the first BGP update message to the time when all received routes are processed.

In Figure 9(a), the bottom line shows the processing

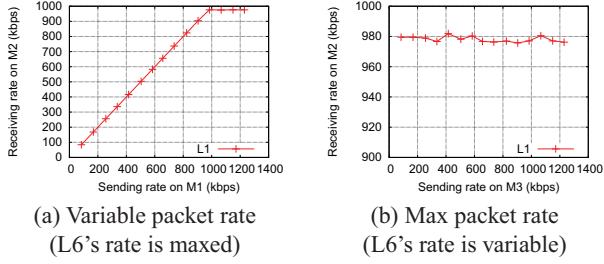


Figure 10: Data plane isolation test.

time of the BGP process on LR1 to process all the routes if LR3 is BGP-inactive. The top line shows the processing time for LR1 when LR3 is also actively processing the BGP message stream. Both processing times increase linearly with the number of routes received. The two lines are almost parallel, meaning that the delay is proportional to the original processing time. The difference of receiving 10k routes is about 13 seconds, 73 seconds for 50k routes. We have verified that the CPU usage is 100% even if only LR1 is BGP-active. We have also used two physical machines to peer with LR1 and LR3 and confirmed that the bottleneck is due to the Juniper router control processor. If these limitations prove to be problematic in practice, solutions exist which allow a hardware separation of logical router control planes [17].

6.3.2 Data plane

L1 and L6 share the same physical interfaces, *eth1* on *Server1* and *ge-0/1/0* on *JuniperRouter1*. We restrict the bandwidth usage of both L1 and L6 to be 1Mbps by applying traffic policer on the ingress interfaces on LR1 and LR3. From the perspective of a given UsrLink, say *L1*, we evaluate two aspects: regardless of the amount of traffic sent on *L6*, (1) *L1* can always achieve the maximum bandwidth allocated (*e.g.*, 1Mbps given a 100Mbps interface); (2) *L1* can always obtain its fair share of the link. To facilitate this test, we apply traffic policer on the ingress interfaces (*ge-0/1/0*) on LR1 and LR3, restricting the bandwidth of L1 and L6 to 1Mbps. Simultaneous traffic is sent from M1 via L1 to M2, and from M3 via L6 to M4.

Figure 10(a) shows the observed receiving rate on M2 (y-axis) as the sending rate of M1 (x-axis) increases, while M3 is sending as fast as possible. The receiving rate matches closely with the sending rate, before reaching the imposed 1Mbps limit. This demonstrates that *L1* capacity is not affected, even if *L6* is maxed out. Figure 10(b) shows the max rate of *L1* can achieve is always around 980kbps no matter how fast *M2* is sending.

6.4 Device failure mitigation

We evaluate the recovery time in response to a hardware failure in ShadowNet. While *Slice1* is running, M1 continuously sends packets to M2 via L1. We then phys-

ically yanked the Ethernet cable on the Ethernet module *ge-0/1/0*, triggering SNMP LinkDown trap message and the subsequent reconfiguration activity. A separate interface (not shown in the figure) is found to be usable, then automatically configured to resurrect the down links. Figure 9(b) shows the packet rate that M2 observes. The downtime is about 7.7 seconds, mostly spent on effecting router configuration change. Failure detection is fast due to continuous SNMP messages, and similarly controller processing takes less than 100ms. This exemplifies the benefit of strong isolation in ShadowNet, as the physical instantiation is dynamically replaced using the previous IP and OSPF configuration, leaving the user perceived slice intact after a short interruption. To further reduce the recovery time, the ShadowNet controller can spread a UsrLink’s instantiation onto multiple physical interfaces, each of which provides a portion of the bandwidth independently.

7 Related work

ShadowNet has much in common with other test/trial networks [10, 27, 22]. However, to our knowledge, ShadowNet is the first platform to exploit recent advances in the capabilities of networking equipment to provide a sharable, composable and programmable infrastructure using carrier-grade equipment running on a production ISP network. This enables a distinct emphasis shift from experimentation/prototyping (enabled by other test networks), to service trial/deployment (enabled by ShadowNet). The fact that ShadowNet utilizes production quality equipment frees us from having to deal with low-level virtualization/partitioning mechanisms, which typically form a significant part of other sharable environments.

A similar service deployment incentive to that espoused by ShadowNet was advocated in [21]. Their service definition is, however, narrower than ShadowNet’s scope which also includes network layer services. Amazon’s EC2 provides a platform for rapid and flexible edge service deployment with a low cost [1]. This platform only rents computing machines with network access, lacking the ability to control the networking aspects of service testing, or indeed network infrastructure of any kind. PLayer [14] is designed to provide a flexible and composable switching layer in data-center environment. It achieves dynamic topology change with low cost; however, it is not based on commodity hardware.

Alimi *et al.* proposed the idea of shadow configuration [8], a new set of configuration files that first run in parallel with existing configuration and then either committed or discarded. The shadow configuration can be evaluated using real traffic load. The downside is that the separation between the production network and the shadowed configuration may not be strongly guaranteed.

This technique requires significant software and hardware modification on proprietary network devices.

We heavily rely on hardware-based and software-based virtualization support [6] in the realization of ShadowNet, for example virtual machines [5] and Juniper's logical router [16]. The isolation between the logical functionality and the physical resource can be deployed to achieve advanced techniques, like router migration in VROOM [26] and virtual machine migration [20, 12], which can be used by ShadowNet.

8 Conclusion

In this paper, we propose an architecture called ShadowNet, designed to accelerate network change in the form of new networks services and sophisticated network operation mechanisms. Its key property is that the infrastructure is connected to, but functionally separated from a production network, thus enabling more realistic service testing. The fact that production-grade devices are used in ShadowNet greatly improves the fidelity and realism achieved. In the design and implementation of ShadowNet, we created strong separation between the user-level representations from the physical-level instantiation, enabling dynamic composition of user-specified topologies, intelligent resource management and transparent failure mitigation. Though ShadowNet currently provides primitives mainly for service testing purposes, as a next step, we seek to broaden the applicability of ShadowNet, in particular, to merge the control framework into the production network for allowing service deployment.

Acknowledgment: We wish to thank our shepherd Jaeyeon Jung as well as the anonymous reviewers for their valuable feedback on this paper.

References

- [1] Amazon Elastic Compute Cloud. <http://aws.amazon.com/ec2/>.
- [2] FEDERICA: Federated E-infrastructure Dedicated to European Researchers Innovating in Computing network Architectures. <http://www.fp7-federica.eu/>.
- [3] GENI: Global Environment for Network Innovations. <http://www.geni.net/>.
- [4] Traffic Control HOWTO. <http://linux-ip.net/articles/Traffic-Control-HOWTO/>.
- [5] VirtualBox. <http://www.virtualbox.org>.
- [6] K. Adams and O. Ageson. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.
- [7] M. Agrawal, S. Bailey, A. Greenberg, J. Pastor, P. Sebos, S. Seshan, K. van der Merwe, and J. Yates. Routerfarm: Towards a dynamic, manageable network edge. SIGCOMM Workshop on Internet Network Management (INM), September 2006.
- [8] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *Proceedings of ACM SIGCOMM*, Seattle, WA, August 2008.
- [9] H. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. Van der Merwe. Anycast CDNs Revisited. 17th International World Wide Web Conference, April 2008.
- [10] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI veritas: realistic and controlled network experimentation. *SIGCOMM Comput. Commun. Rev.*, 36(4):3–14, 2006.
- [11] Cisco Systems. MPLS VPN Carrier Supporting Carrier. http://www.cisco.com/en/US/docs/ios/12_ost/12_ost14/feature/guide/csc.html.
- [12] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005.
- [13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.
- [14] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4), 2008.
- [15] Juniper Networks. Configuring Interprovider and Carrier-of-Carriers VPNs. <http://www.juniper.net/>.
- [16] Juniper Networks. Juniper Logical Routers. <http://www.juniper.net/techpubs/software/junos/junos85/feature-guide-85/id-11139212.html>.
- [17] Juniper Networks. Juniper Networks JCS 1200 Control System Chassis. <http://www.juniper.net/products/tseries/100218.pdf>.
- [18] Juniper Networks. Juniper Partner Solution Development Platform. <http://www.juniper.net/partners/osdp.html>.
- [19] Juniper Networks. JUNOS 9.2 Policy Framework Configuration Guide. <http://www.juniper.net/techpubs/software/junos92/swconfig-policy/frameset.html>.
- [20] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.
- [21] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology Into the Internet. In *Proc. of ACM HotNets*, 2002.
- [22] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006.
- [23] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. Proc. of ACM HotNets, 2004.
- [24] J. Turner and N. McKeown. Can overlay hosting services make ip ossification irrelevant? PRESTO: Workshop on Programmable Routers for the Extensible Services of TOMorrow, May 2007.
- [25] J. E. Van der Merwe et al. Dynamic Connectivity Management with an Intelligent Route Service Control Point. Proceedings of ACM SIGCOMM INM, October 2006.
- [26] Y. Wang, E. Keller, B. Biskborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: live router migration as a network-management primitive. *SIGCOMM Comput. Commun. Rev.*, 38(4), 2008.
- [27] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

Design and implementation of TCP data probes for reliable and metric-rich network path monitoring

Xiapu Luo, Edmond W. W. Chan, and Rocky K. C. Chang

The Hong Kong Polytechnic University, Hong Kong

{csxluo|cswwchan|csrchang}@comp.polyu.edu.hk

Abstract

Monitoring network services and diagnosing their problems often require active probing methods. Current probing methods, however, are becoming unreliable, because of interferences from various middleboxes, and inadequate due to their limited path metrics support. In this paper, we present the design and implementation of OneProbe, a new TCP probing method for reliable and metric-rich path monitoring. We have implemented HTTP/OneProbe (i.e., OneProbe for HTTP) which sends TCP data probes containing legitimate HTTP requests to induce HTTP responses for path measurement. Since the probing method is based on TCP's basic data transmission mechanisms, OneProbe works correctly on all major operating systems and web server software, and on 93% of the 37,874 websites randomly selected from the Internet. We also successfully deployed HTTP/OneProbe to monitor a number of network paths for over a month and obtained interesting and useful measurement results.

1 Introduction

The ability of measuring a network path's quality is important for monitoring service level agreement, choosing the best route, diagnosing performance problems, and many others. This paper considers active measurement methods that do not require the remote endpoint's cooperation in terms of setting up additional software. A non-cooperative method therefore measures the path quality solely based on the response packets induced by its probes. Compared with cooperative methods, non-cooperative methods offer the potential advantage of monitoring a large number of paths from a single system.

The design and implementation of a reliable non-cooperative method is very challenging for the Internet landscape today. A main challenge is to obtain reliable measurement in the midst of interferences from various middleboxes. By reliability, we mean three specific requirements. First, the method can always induce the expected response packets from the remote endpoints, re-

gardless of their operating systems, for path measurement. Second, the method can measure the path quality experienced by data packets. Third, the method can support an adequate sampling rate and sound sampling patterns for obtaining reliable measurement samples.

However, the most practiced measurement methods are not reliable according to our definition. Most notably, routers and end hosts do not always respond to ICMP Ping and Traceroute [24]. Even when ICMP packets are returned, the Ping measurement results may not be trustworthy [38], because the ICMP packets and TCP data packets are processed on different paths in routers. The same can also be said for the probe and response packets that are TCP SYNs, TCP RSTs, and TCP ACKs (pure TCP acknowledgment packets). Other middleboxes, such as accelerators, traffic shapers, load balancers, and intrusion detection systems, can further increase the measurement inaccuracy. A related problem is that their sampling rates cannot be too high.

Another motivation for this work is that an existing non-cooperative method usually supports a very limited number of path metrics. As the quality expected from network paths could be different for various applications, it is necessary to measure the path quality using as many metrics as possible. There are three specific shortcomings responsible for the current limitation. First, many methods, such as Ping, can only measure round-trip path quality. Second, almost all methods (with the exception of tulip [26]) only support one or two types of metrics (e.g., sting [34] for packet loss and POINTER [25] for packet reordering). Third, all methods cannot measure path metrics with different response packet sizes (e.g., sting measures reverse-path packet loss using only TCP ACKs).

1.1 A new non-cooperative measurement approach

Our approach to tackling the reliable path monitoring problem is to conduct measurement in a legitimate TCP application session and to use TCP data packets for the

probe and response packets. We avoid using the TCP ACKs returned from the remote endpoints for measurement, because some systems do not return them. Moreover, TCP ACKs are not reliable, and their packet size cannot be changed. Using TCP data packets for the probe and response packets resolves all three problems.

The new TCP data probe is also capable of measuring multiple path metrics—round-trip time (RTT), forward-path and reverse-path packet loss rates, and forward-path and reverse-path packet reordering rates—all at the same time from the same probe. Therefore, we call this new TCP probing method OneProbe: the same probe for multiple path metrics. Although tulip also measures multiple metrics, it cannot measure some packet loss scenarios [26]. Moreover, the tulip probes are different for loss and reordering measurement.

We have implemented HTTP/OneProbe (i.e., OneProbe for HTTP/1.1) which sends legitimate HTTP GET requests in the TCP data probes to induce HTTP response messages for path measurement. Our validation results have shown that the TCP data probes work correctly on all major operating systems and web server software. It also worked on 93% of the 37,874 websites randomly selected from the Internet. We have also enhanced the basic HTTP/OneProbe by using concurrent TCP connections and TCP timestamps option, and improving the process of obtaining sufficient HTTP responses for continuous measurement.

TCP Sidecar [35, 36], a measurement platform based on TCP, is closest to our work regarding the requirement of evading middleboxes’ interferences. TCP Sidecar’s approach is to inject probes into an externally generated TCP flow. Since the focus of TCP Sidecar is to provide a platform for unobtrusive measurement, it does not provide a new probing method to its “passengers.” OneProbe, on the other hand, establishes a new TCP flow for measurement and customizes TCP data probes for measuring multiple path metrics.

1.2 Contributions of this work

1. This paper explains why the existing non-cooperative measurement methods are becoming unreliable and inadequate for the Internet today and proposes to use TCP data probes for reliable and metric-rich path measurement.
2. This paper proposes a new TCP probing method called OneProbe which sends two TCP data packets to measure multiple path metrics. The correctness of the probe responses was validated on operating systems, web server software, and websites.
3. This paper describes the implementation details of HTTP/OneProbe, such as the method of obtaining suitable http URLs for measurement and using HTTP/1.1’s request pipelining to facilitate continu-

ous measurement in a persistent HTTP connection.

4. This paper prescribes three enhancements to the basic HTTP/OneProbe: improving the process of inducing HTTP responses, using TCP timestamps option to enhance the measurement, and employing concurrent TCP connections to support a higher sampling rate and different sampling patterns.
5. This paper presents testbed experiment results for evaluating HTTP/OneProbe’s performance and measurement accuracy, and our measurement experience of monitoring network paths for over a month using HTTP/OneProbe and other tools.

2 Related work

Since OneProbe measures RTT, packet loss, and packet reordering in an legitimate TCP session, it is mostly related to several non-cooperative measurement tools: sting, POINTER, tulip [26], and TCP sidecar.

OneProbe overcomes sting’s two main shortcomings for loss-only measurement: unreliability due to anomalous probe traffic and a lack of support for variable response packet size. The probe packets in sting may be filtered due to their highly unusual patterns (a burst of out-of-ordered TCP probes with zero advertised window). The reverse-path loss measurement based on TCP ACKs may be under-estimated for a larger packet size [15]. We recently evaluated sting on the set of 37,874 websites with the two probe packet sizes considered in [34]. With the 41-byte probes, the sting measurement was unsuccessful for 54.8% of the servers; the non-success rate for the 1052-byte probes was even close to 100%.

OneProbe overcomes POINTER’s two similar shortcomings for reordering-only measurement. The first two POINTER methods (ACM and SAM1) send TCP probe packets with unacceptable acknowledgment numbers (ANs) and sequence number (SNs) to induce TCP ACKs for measurement. Therefore, the probes could be considered anomalous, and the response packet size cannot be changed. The third method (SAM2), on the other hand, sends probes with acceptable SNs but the ANs become unacceptable if the probe packets are reordered.

Tulip, being a hop-by-hop measurement tool, was designed to localize packet loss and reordering events on network paths, and to measure queueing delay. Tulip’s loss and reordering measurement, however, is based on the unwarranted assumption that the remote hosts and routers support consecutive IPID (IP’s identification) values. We tested tulip using the same set of web servers for sting. In our experiments, tulip measured the last hops of the paths. The tests were unsuccessful for 80% of the servers for loss and reordering measurement—50% of them failed to respond to tulip’s UDP probes, and another 30% failed to return consecutive IPID values.

TCP Sidecar provides support for injecting measure-

ment probes in a non-measurement TCP connection. The probes are limited to TCP ACKs and replayed TCP data packets, because they must not interfere with the normal data transmissions in the TCP connection. As a result, the probes do not measure all packet loss scenarios and packet reordering. Due to the same reason, the sampling pattern and rate cannot be controlled, because a probe is sent only after the connection is idle for some time (e.g., 500 milliseconds in [36]).

3 OneProbe

OneProbe is a new probing method operating at the TCP layer. Each probe consists of two customized TCP data packets to induce at most two new TCP data packets from the remote endpoint for path measurement. Moreover, the probe and response packets carry legitimate application data, so that the remote side will perceive the probe traffic as coming from a legitimate application session. In a client-server application protocol, the probes usually carry application requests, and the response packets contain the requested objects. Therefore, an OneProbe implementation comprises two main components: OneProbe and a TCP application-dependent component.

OneProbe can be implemented for any TCP application protocol that provides support for requesting data from the remote endpoint. This paper presents HTTP/OneProbe (HTTP/OP in short), an OneProbe implementation for HTTP/1.1 [33]. Figure 1 shows the main components of HTTP/OP. An HTTP/OP user inputs an http URL, and the probe and response packet sizes (measured in terms of the IP packet size). The HTTP helper, an application-dependent component, first comes up a set of qualified URLs for the specified packet sizes and then prepares the corresponding HTTP GET messages. The user may also specify the sampling pattern and rate which, together with the HTTP GET messages, are used for OneProbe measurement at the TCP layer.

3.1 The probe design

The probe is the result of several design choices. The first advantage of using TCP probes (instead of application-layer probes) is that the same probing mechanism could be implemented for many TCP application protocols. TCP probes can also provide more accurate measurement about the network path quality than higher-layer probes. Moreover, using two packets is a minimum requirement for packet reordering measurement. For loss measurement, the second packet can help determine where—the forward path (from OneProbe to the remote endpoint) or the reverse path—the first packet is lost.

Another key issue in the probe design is what kind of response packets to induce from the remote endpoint. To measure the reverse-path quality with the same types of metrics, the probe is designed to induce at most two new

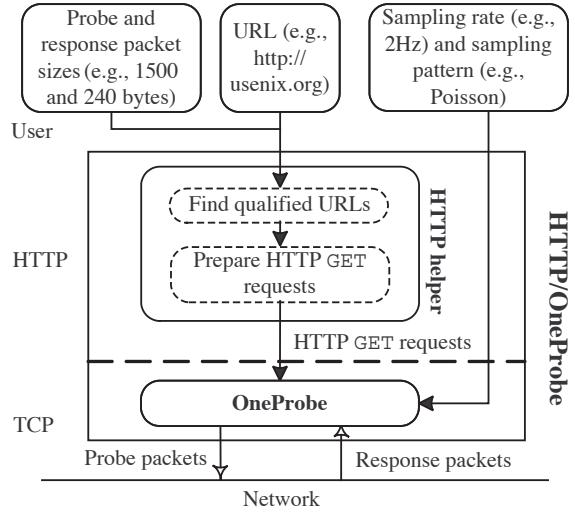


Figure 1: The main components of HTTP/OneProbe.

TCP data packets from the remote endpoint. These two response packets are used for measuring the reverse-path quality in a similar way as the two probe packets for the forward-path quality. Furthermore, the response packets are distinguishable for almost all possible delivery statuses of the probe and response packets. As a result, OneProbe can measure both forward-path and reverse-path quality primarily based on the response packets.

3.2 The probing process

HTTP/OP sends a sequence of probes in a persistent HTTP connection (over a single TCP connection). Each probe packet contains a legitimate HTTP request, and each response packet contains legitimate data requested by HTTP/OP. To focus on the probing process in this section, we temporarily ignore the application-level issues and assume that the TCP server always has enough application data to send back to HTTP/OP. We also postpone the explanation on how OneProbe can set the user-specified packet sizes to section 4.

We use Figure 2 to explain the probing process. Denote a probe packet by $Cm|n$ and a response packet by $Sm|n$. Both packets are TCP data packets, and m and n are the TCP data segment's SN and AN, respectively. All the TCP data segments considered in this paper are of full size (i.e., the maximum segment size, MSS). Therefore, we simply use $m = 1, 2, \dots$ to enumerate the server's TCP data segments and $1', 2', \dots$ OneProbe's TCP data segments. For example, OneProbe sends its fourth data segment in $C4'|2$ that also acknowledges the first two data segments from the server. Moreover, when the AN is not important, we just use Cm and Sm .

OneProbe customizes and dispatches the successive probes according to the following three rules:

- P1. (Dispatching a new probe) A new probe is dispatched only after receiving two new data segments from the server and the acknowledgment for the data segments in the probe.
- P2. (Acknowledging one data segment) Each probe packet acknowledges *only one* data segment from the server, although both have been received by the time of sending the first probe packet.
- P3. (Controlling the send window size) The probe packets advertise a TCP receive window of two segments in an attempt to constrain the server's TCP send window size to two segments.

Figure 2 depicts two successive probe rounds (the first round denoted by dotted lines and the second by solid lines). According to P1, OneProbe sends a new probe of $\{C3'|1, C4'|2\}$ (for a new probe round) after receiving $S1|1'$ and $S2|2'$. Therefore, the packet transmissions in the first round do not overlap with that in the next. Moreover, if the server's congestion window size ($cwnd$) is at least two segments, P3 will ensure that its send window size is set to two segments. Finally, based on P2 and P3, the server can send only one new data segment after receiving a probe packet if the probe packets are received in the original order.

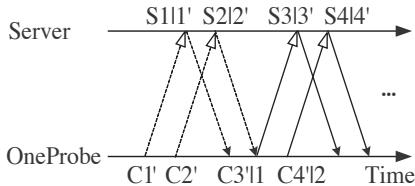


Figure 2: Two successive probe rounds in OneProbe.

Although OneProbe manipulates the TCP packet transmissions according to P1-P3, there are no apparent anomalies existing in the probe packets. It only appears to the server that the client has a low receive buffer, and its send window is always full. Moreover, according to our measurement experience, the OneProbe transmission pattern was not construed for an anomalous TCP flow. We received only a couple of complaints about our measurement activities for the past two years; one of them came from a website that normally received very few external requests.

3.3 Measuring RTT

OneProbe measures the RTT based on a probe packet and its induced new data packet (e.g., $C3'|1$ and $S3|3'$ in Figure 2). Therefore, in the absence of packet loss, OneProbe normally obtains two RTT observations in a probe round. However, OneProbe uses only the first-probe-packet-RTT for measurement, because the second probe packet's RTT may be biased by the first packet [10].

3.4 Detecting packet loss and reordering events

There are five possible path events regarding the two probe packets on the forward path:

- F0. Both probe packets arrive at the server with the same order.
- FR. Both probe packets arrive at the server with a reverse order.
- F1. The first probe packet is lost, but the second arrives at the server.
- F2. The first probe packet arrives at the server, but the second is lost.
- F3. Both probe packets are lost.

There are also five similar events for the two new response packets on the reverse path: R0, RR, R1, R2, and R3 (by replacing “probe” with “response” and “the server” with “OneProbe” in the list above). As a result, there are 18 possible loss-reordering events, as shown in Table 1: the 17 events indicated \checkmark and one event for F3 (there is no \checkmark , because this is a forward-path-only event). Others indicated by – are obviously not possible.

Table 1: The 18 possible loss-reordering events for the two probe packets and two response packets.

	R0	RR	R1	R2	R3
F0	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
FR	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
F1	\checkmark	\checkmark	\checkmark	\checkmark	\checkmark
F2	\checkmark	–	\checkmark	–	–
F3	–	–	–	–	–

OneProbe can detect almost all the 18 path events based on the response packets. Considering the $\{C3'|1, C4'|2\}$ probe in Figure 2, Table 2 summarizes the response packets induced for the 18 cases based on RFC 793 [20]. In addition to the new data segments 3 and 4, the server may retransmit old data segments 1, 2, and 3, and we use $\hat{S}m|n$ to refer to a data retransmission. Since the server responses are based on TCP's two basic mechanisms: acknowledgment-clocked transmissions and timeout-based retransmissions, all operating systems are expected to produce the same responses.

Figure 2 has already illustrated the event $F0 \times R0$; Figure 3 ($C1'$ and $C2'$ are omitted) illustrates four other cases: $FR \times R0$, $F1 \times R0$, $F2 \times R0$, and $F3$. The rest can be easily constructed from the illustrations for these five events. Note that, because of P1, the server retransmits old data segments in all four cases. The main purpose for withholding a new probe, even after receiving two new data segments (e.g., in the events $FR \times R0$ and $F1 \times R0$), is to induce retransmissions for path event differentiation.

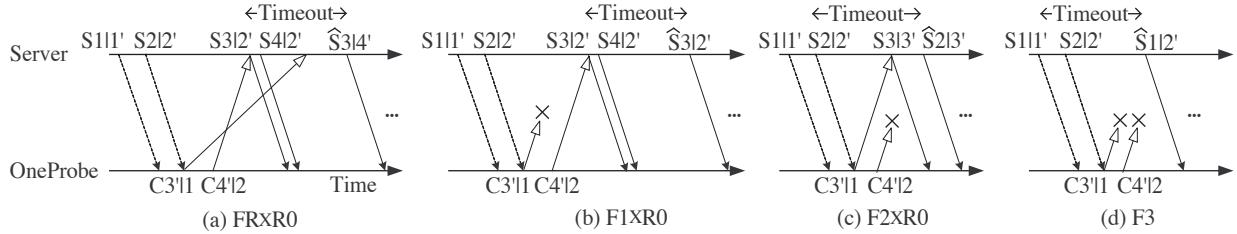


Figure 3: OneProbe’s packet transmissions for the path events $\text{FR} \times \text{R}0$, $\text{F}1 \times \text{R}0$, $\text{F}2 \times \text{R}0$, and $\text{F}3 \times \text{R}0$.

Table 2: The response packets induced by the $\{C3'|1, C4'|2\}$ probe for the 18 path events according to RFC 793.

Path events	1st response packets	2nd response packets	3rd response packets
1. $\text{F}0 \times \text{R}0$	$S3 3'$	$S4 4'$	–
2. $\text{F}0 \times \text{RR}$	$S4 4'$	$S3 3'$	–
3. $\text{F}0 \times \text{R}1$	$S4 4'$	$\hat{S}3 4'$	–
4. $\text{F}0 \times \text{R}2$	$S3 3'$	$\hat{S}3 4'$	–
5. $\text{F}0 \times \text{R}3$	$\hat{S}3 4'$	–	–
6. $\text{F}R \times \text{R}0$	$S3 2'$	$S4 2'$	$\hat{S}3 4'$
7. $\text{F}R \times \text{RR}$	$S4 2'$	$S3 2'$	$\hat{S}3 4'$
8. $\text{F}R \times \text{R}1$	$S4 2'$	$\hat{S}3 4'$	–
9. $\text{F}R \times \text{R}2$	$S3 2'$	$\hat{S}3 4'$	–
10. $\text{F}R \times \text{R}3$	$\hat{S}3 4'$	–	–
11. $\text{F}1 \times \text{R}0$	$S3 2'$	$S4 2'$	$\hat{S}3 2'$
12. $\text{F}1 \times \text{RR}$	$S4 2'$	$S3 2'$	$\hat{S}3 2'$
13. $\text{F}1 \times \text{R}1$	$S4 2'$	$\hat{S}3 2'$	–
14. $\text{F}1 \times \text{R}2$	$S3 2'$	$\hat{S}3 2'$	–
15. $\text{F}1 \times \text{R}3$	$\hat{S}3 2'$	–	–
16. $\text{F}2 \times \text{R}0$	$S3 3'$	$\hat{S}2 3'$	–
17. $\text{F}2 \times \text{R}1$	$\hat{S}2 3'$	–	–
18. $\text{F}3$	$\hat{S}1 2'$	–	–

3.4.1 Distinguishability of the path events

The different combinations of the SN and AN in the response packets enable OneProbe to distinguish almost all the 18 path events. It is not difficult to see, by sorting Table 2 according to the three response packets, that each sequence of the response packets matches uniquely to a path event, except for the following three cases:

- A1. $\text{F}1 \times \text{R}2$ and $\text{F}1 \times \text{R}3$: These two events cannot be distinguished based on the response packets, because $S3|2'$ and $\hat{S}3|2'$ are identical, and the server may retransmit more than once.
- A2. $\text{F}1 \times \text{RR}$ and $\text{F}1 \times \text{R}1$: The reasons for their indistinguishability are similar to that for A1.
- A3. $\text{F}0 \times \text{R}3$ and $\text{F}R \times \text{R}3$: Both events have the same response packet $\hat{S}3|4'$.

The ambiguities in A1 and A2 make the delivery status of $S3|2'$ uncertain. The ambiguity in A3, on the other hand, makes the probe’s order of arrival uncertain. Our current implementation disambiguates A1 and A2 by measuring the time required for $S3|2'$ (or $\hat{S}3|2'$) to arrive. It usually takes a much longer time to receive

$\hat{S}3|2'$, the retransmission of $S3|2'$.

3.5 Assistance from TCP ACKs

Recall that an important design choice for OneProbe is not to rely on TCP ACKs. However, some ACKs, if received by OneProbe, can assist in detecting the path events. There are two such ACKs: out-of-ordered-packet ACK (OOP-ACK) and filling-a-hole ACK (FAH-ACK). Referring to Figure 3(a), the early arrival of $C4'|2$ could immediately trigger an OOP-ACK, whereas the late arrival of $C3'|1$ could immediately trigger an FAH-ACK. According to our measurement, some systems did not return the OOP-ACK, but all the systems tested returned the FAH-ACK.

Even though the system responses regarding the FAH-ACK are uniform, OneProbe still does not rely on it for measurement, because it could be lost. Instead, OneProbe exploits these ACKs, if received, to enhance its measurement. The first is using the FAH-ACK to accelerate the detection of the forward-path reordering events (i.e., $\text{FR} \times *$) without waiting for the data retransmissions. The second is using the FAH-ACK to disambiguate A3 that is the only unresolved case. An arrival of FAH-ACK, in addition to $\hat{S}3|4'$, clearly signals an $\text{FR} \times \text{R}3$ event.

3.6 Starting a new probe round

Out of the 18 path events, only the path events 1-2 fulfill the conditions for dispatching a new probe in P1 immediately after receiving two response packets. Moreover, path events 3 and 6-8 fulfill the conditions immediately after receiving a data retransmission. However, the condition is not met for the rest (i.e., events 4-5 and 9-18). Another related problem is that the server’s cwnd is dropped to one segment for all the path events that involve timeout-based retransmissions (i.e., path events 3-18).

To address the two problems that prevent OneProbe from starting a new probe round, OneProbe will first send one or more new TCP ACKs to increase the server’s cwnd back to two for path events 3-18. After receiving two new data segments, OneProbe dispatches a new probe: $\{C5', C6'\}$ for events 3-10, $\{C4', C5'\}$ for events 16-17, and $\{C3', C4'\}$ for event 18. Handling events 11-15 is more complicated. If a new probe of

$\{C3', C4'\}$ were used, the server will drop $C4'$, because it has already been received. The current implementation restarts the connection when encountering these path events. A better approach is to retransmit $C3'$ with the respective ANs and to use a new probe of $\{C5', C6'\}$.

3.7 Sampling the packet loss and reordering rates

OneProbe samples the packet loss and reordering rates from consecutive probe rounds. Similar to the RTT measurement, OneProbe uses only the first packet for the loss measurement. After conducting a number of consecutive probe rounds, say 120, over one minute, OneProbe computes the forward-path (and reverse-path) loss rate by dividing the number of the first-probe-packet-loss events (and the first-response-packet-loss events) by 120. OneProbe computes the packet reordering rates in a similar manner.

4 HTTP/OneProbe

We have implemented HTTP/OP as a user-level tool (around 8000 lines of C code) on unmodified Linux 2.6 kernel. As shown in Figure 1, HTTP/OP consists of two main components: HTTP helper and OneProbe. HTTP helper handles the issues concerning the HTTP 1.1 protocol, whereas OneProbe implements OneProbe. This section considers a basic HTTP/OP that utilizes a persistent HTTP/1.1 connection.

4.1 The HTTP helper

The HTTP helper's main tasks include finding one or more *qualified http URLs* for the user-specified packet sizes and preparing the HTTP GET requests for them. Figure 4 shows that HTTP/OP sends an initial HTTP GET request for a qualified url-1 in $C0'$. The server replies with an HTTP response message sent in $S1, S2, \dots$. HTTP/OP also sends the same request in all subsequent probe packets. Note that before sending the first probe $\{C1', C2'\}$, HTTP/OP sends an ACK to ramp up the server's cwnd to two segments. Therefore, $C0'$ and $S1-S3$ are not used for OneProbe measurement.

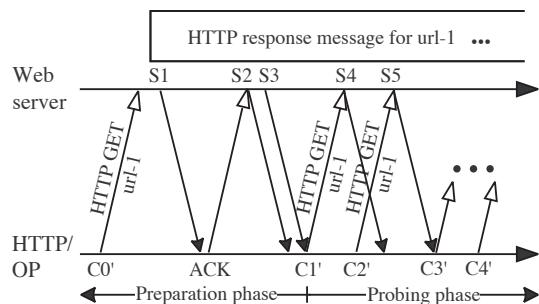


Figure 4: HTTP/OP sends HTTP GET requests for url-1 for OneProbe measurement.

4.1.1 Finding qualified http URLs

An http URL is considered qualified if its HTTP GET request can be retrofitted into a probe packet, and the GET request can induce at least five response packets from the server. A minimum of five response packets is required because of the three additional response packets for the cwnd ramp-up. Let Z_p and Z_r be the user-specified probe packet size and response packet size, respectively. Therefore, the length of the HTTP GET request for a qualified URL will not exceed $Z_p - 40$ bytes (assuming a 40-byte TCP/IP header). Moreover, the length of corresponding HTTP response message, including the response header and message body, must be at least $5 \times (Z_r - 40)$ bytes.

Checking the length of the GET request is simple. Verifying whether a user-specified URL meets the size requirement for the response packets, however, requires some work. If the Content-Length header field is present in the HTTP response message, the length is just a sum of the field value and the response header's length. Otherwise, the helper will download the entire HTTP response message to determine the length. If no qualified URL can be obtained, the helper will perform web crawling to retrieve all the available URLs, starting at the root of the web server and down to a certain depth level (five for our case). Our implementation for the web crawling process is based on the recursive retrieval technique implemented in Wget [17].

Besides, the HTTP GET request for a qualified URL must induce a 200 (OK) response. We purposely do not use those with 404 (Not Found) responses in order not to cause security alerts on the site. We also avoid using HTTP response messages that do not have a message body (e.g., 304 (Not Modified)).

4.1.2 Preparing the HTTP GET requests

To craft a Z_p -byte probe packet for an HTTP request, the helper expands the packet size through the Referer field. Since some web servers only accept requests referred from their own web pages, the helper first appends the requested URL to the Referer field to avoid blocking. If the packet size still falls short, the helper further appends a customized string consisting of a probe ID and an email address for our project (for lodging a complaint [18]) repeatedly until reaching the packet size. Moreover, to reduce the delay in dispatching the probes due to possible context switching, the HTTP helper has prepared the HTTP GET requests for the qualified http URLs before starting the OneProbe measurement.

HTTP/OP exploits HTTP/1.1's request pipelining to include a GET message in each probe packet for path measurement. The pipelined HTTP GET requests could be for a single or multiple URLs. There are also other alternatives, such as sending a large GET message in sev-

eral probe packets or including multiple GET messages in a probe packet. But we did not adopt them, because they are either delaying the return of the response packets or introducing too many request messages.

Moreover, an HTTP response message usually will not fully occupy the last response packet. Therefore, a full-sized response packet may contain data from two HTTP response messages. However, we have also observed that some response packets are not full-sized packets, because they contain only the last chunks of the response messages. Our current implementation will close the connection whenever detecting a small segment and ignore the probe rounds involving small segments. A better approach is perhaps to continue the next probe round using the next HTTP response message in the same connection.

4.2 An implementation of OneProbe

OneProbe manages the measurement in two levels: session and TCP connection. An OneProbe session could involve concurrent TCP connections (see section 5.3 for this enhancement). Figure 5 shows OneProbe’s main tasks for a TCP connection in two consecutive phases: *preparation* and *probing*. The preparation phase is for performing the ground works for the probing phase. In the probing phase, OneProbe dispatches the probes containing the HTTP GET requests that have been prepared by the HTTP helper, analyzes the response packets, and terminates the connection when the session ends or encounters exceptions. OneProbe also includes a diagnosis module to detect self-induced packet losses.

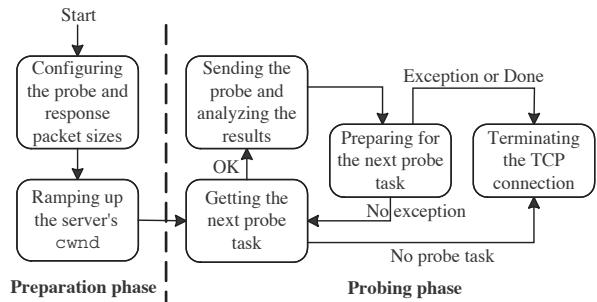


Figure 5: OneProbe’s major tasks in the preparation and probing phases for a TCP connection.

4.2.1 Session management

There are two mains tasks in the session management. The first task is that OneProbe establishes and maintains a system-configurable number of TCP connections for a measurement session (one connection for the basic HTTP/OP). As a second task, OneProbe prepares a *probe schedule* according to the user-specified sampling pattern and rate before starting the measurement. The

schedule contains a list of *probe tasks*, each of which includes a dispatch time and a probe number. The probe tasks are enqueued to a *probe-schedule queue* as soon as they are generated. OneProbe currently supports periodic and Poisson sampling, and it is not difficult to admit others. For the Poisson sampling, our implementation is based on the method 3 in RFC 2330 [31] which eliminates possible timing errors in dispatching the probes.

4.2.2 The preparation phase

OneProbe configures the probe and response packet sizes during the TCP three-way handshake. OneProbe advertises its MSS (say MSS_c) in the TCP SYN segment to control the size of the server’s response packets. From the TCP SYN/ACK segment returned by the server, OneProbe learns the server’s advertised MSS (say MSS_s). As a result, Z_p must be less than $MSS_s + 40$ bytes, and $Z_r = \min\{MSS_c, MSS_s\} + 40$ bytes. Therefore, OneProbe can dictate the server’s response packet size by advertising an $MSS_c < MSS_s$.

Another purpose of this phase, as already shown in Figure 4, is to ramp up the server’s cwnd to two segments for starting the first probe round. If the server’s initial cwnd is at least two segments (detected by receiving two response packets after the initial HTTP GET request), then the first probe round can be started without sending the ACK.

4.2.3 The probing phase

Preparing for the probes The probing phase starts as soon as receiving two response packets from the server (see Figure 4). To dispatch a probe, OneProbe first retrieves a probe task from the probe-schedule queue. Moreover, any slipped probe task, for which its dispatch time has already passed the current time, will be removed from the queue and discarded. When the probe schedule is empty, OneProbe closes the TCP connection.

After obtaining a probe task, OneProbe uses `clock_nanosleep()` in `time.h` to perform a high-resolution sleep until reaching the dispatch time. Upon waking up, OneProbe draws a pair of HTTP GET requests randomly from the list of the GET requests already prepared by the HTTP helper and sends each in a probe packet. To ensure a successful delivery of the probe to the network, OneProbe captures each dispatched probe packet using `libpcap`.

Dispatching the probes Similar to other measuring systems, such as Scriptroute [37], we have used Linux raw socket to craft and send the probe packets, and the `libpcap` 1.0.0 library to capture the probe and response packets. As a result of bypassing Linux’s normal TCP/IP processing, the kernel is unaware of OneProbe’s TCP connections and will therefore respond with a TCP RST for each response packet received. Our implementation

blocks the RST traffic using Linux’s iptables.

Another important issue is to timestamp each probe and response packet accurately for the RTT measurement. Since we have already used libpcap to capture packets, we use the timestamp from the pcap_pkthdr structure of each probe and response packet to measure the RTT with microsecond resolution. An alternative is to use the recently proposed TSC clock [14] that provides a highly accurate timing information through the kernel timestamping, but accessing it requires a kernel patch. The user-level timestamp from gettimeofday(), on the other hand, is unreliable, because its accuracy can be affected by system’s context switching.

Analyzing the response packets OneProbe captures the response packets (and probe packets) using libpcap and writes all the captured packets to a dump file (which can be opened by pcap_dump_offline() available in the libpcap library) for analysis. OneProbe determines the path event based on the sequence of response packets in Table 2 and the assistance of TCP ACKs discussed in section 3.5. It also measures the first-probe-packet-RTT from the packet timestamps. In processing the response packets, OneProbe also filters packets irrelevant to the measurement, such as TCP window updates. Furthermore, OneProbe computes from a consecutive number of probe rounds the statistical metrics (in terms of, e.g., mean and median) for the RTT, loss rates, and reordering rates.

OneProbe supports both online and offline processing of the response packets. The online processing is possible, because OneProbe only needs to identify the TCP data packet received from the server. However, we have set the default processing to offline mainly for preventing the processing workload from influencing the probing process. Another advantage of the offline approach is to facilitate a more accurate (as compared with the online approach) disambiguation of A1 and A2 based on the RTT samples collected in the measurement (as discussed in section 3.4).

4.2.4 Diagnosing self-induced packet losses

OneProbe performs a self-diagnosis to confirm that the measurement is free of self-induced packet losses. For the forward-path measurement, failures of sending out the probe packets are still possible, despite that the implementation always validates the successful invocation of the sendto() function. To detect these self-induced losses, OneProbe uses libpcap to verify the delivery of each outgoing probe packet to the network. For the reverse-path measurement, self-induced losses could also occur to the response packets due to insufficient buffer space. OneProbe monitors the ps_drop variable returned by the libpcap’s pcap_stats() function to detect such losses.

5 Enhancements

This section describes three enhancements to the basic HTTP/OP presented in the last section. The first enhancement is to improve the process of inducing sufficient HTTP responses. We have implemented additional mechanisms to prevent web servers from compressing the requested objects and to use unqualified URLs for measurement. The second is to disambiguate A3 using TCP timestamps option. The third enhancement is using multiple TCP connections in a measurement session to satisfy the user-specified sampling rate and pattern. With a single TCP connection, the sampling rate is constrained to at most one per RTT, and the RTT variations also make it difficult to realize the user-specified sampling pattern.

5.1 Improving the HTTP response solicitation

Avoiding message compression The first improvement is to prevent web servers from compressing HTTP responses which, for example, is performed by Apache server’s mod_deflate module [1]. The compressed responses could affect OneProbe measurement, because the expected number of response packets for a qualified URL may be reduced. Therefore, each HTTP GET request specifies Accept-Encoding: identity;q=1, *;q=0, where identity;q=1 indicates that the identity encoding (i.e., no transformation) should be performed on the entity of the response, and *;q=0 means avoiding other encoding methods.

Using unqualified URLs for measurement As a second improvement, HTTP/OP exploits the range request feature in HTTP/1.1 to use unqualified URLs for path measurement. A range request can be used to request multiple overlapped ranges of the same web object from a web server that accepts range requests. Therefore, even an unqualified URL can be “expanded” to fulfill the minimum size requirement for the response packet.

We have implemented this enhancement in the HTTP helper which can verify whether the server supports the range request via the Accept-Ranges header field in the HTTP response message. If the HTTP helper cannot find any qualified URL but discover that the server supports the range request feature, it will craft a range request as discussed above to induce HTTP response messages for OneProbe measurement.

5.2 Using TCP timestamps to disambiguate A3

In addition to the FAH-ACK, we have proposed and implemented a method to disambiguate A3 using the TCP timestamps option [21]. In this enhancement, each probe packet contains a distinct timestamp in the TCP option field. If the server also supports the TCP timestamps option, it will retain the timestamp received from the *most recent* probe packet that advances its receive window and

echo it in its next response packet. Therefore, the server retains $C4'$'s timestamp for the case of $F0 \times R3$ and $C3'$'s timestamp for the case of $FR \times R3$. As a result, the two path events can be distinguished based on the different timestamps in $\hat{S}3|4'$.

5.3 Using multiple TCP connections

To extend the basic HTTP/OP to using N TCP connections, we have used the POSIX Threads (*pthreads*) library to create and manage multiple threads. A single thread is used for managing the measurement session, and N worker threads are created for managing the TCP connections separately. OneProbe also monitors the health of the connections to ensure that there are always N TCP connections available throughout the measurement session.

Since some web servers may limit the number of concurrent TCP connections initiated from an IP address, OneProbe assigns randomly selected source IP addresses from an address pool to the N connections. Our experience shows that $N = 10$ is sufficient for supporting periodic sampling with a rate of two probes per second. A higher N , however, is expected for Poisson sampling because of the high inter-probe delay variability.

6 Evaluation

This section presents three sets of evaluation results. The first one evaluates whether different systems and web servers respond to OneProbe's probes correctly. The second evaluates how the latency induced by web servers will affect the accuracy of the HTTP/OP measurement. The final set evaluates the effect of the HTTP/OP measurement on the system resource consumption in the measuring system and web servers.

6.1 Validation of OneProbe

We have designed a small, but just sufficient, suite of validation tests (called Validator) for OneProbe. A system or web server that passes all the tests can be used by OneProbe for path measurement. Table 3 describes the four validation tests V0-V2 that “simulate” the forward-path events F0-F2, respectively. Same as OneProbe, Validator constrains the server's cwnd to two segments. Moreover, Validator does not acknowledge the response data packets in order to simulate reverse-path losses. Therefore, the data retransmissions are expected to be the same as in Table 2. Note that these tests for reverse-path losses have already covered the test for F3, because withholding the next probe is the same as losing it.

6.1.1 Results for operating systems and web servers

We applied Validator to test the major operating systems and web server software listed in Table 4. Three trials were performed for each system and server. A test was considered successful if all four validation tests were

Table 3: A suite of four validation tests performed by Validator.

Tests	Testing probes	Expected packets induced from server	Expected data retransmissions
V0.	$\{C3', C4'\}$	$\{S3 3', S4 4'\}$	$\hat{S}3 4'$
VR.	$\{C4', C3'\}$	$\{S3 2', S4 2'\}$	$\hat{S}3 4'$
V1.	$C4'$ only	$\{S3 2', S4 2'\}$	$\hat{S}3 2'$
V2.	$C3'$ only	$S3 3'$	$\hat{S}2 3'$

passed in at least one trial. The validation results were all successful.

Table 4: The 39 systems and 35 web server software that passed the OneProbe validation tests.

Systems tested in our lab.:	FreeBSD v4.5/4.11/5.5/6.0/6.2, Linux kernel v2.4.20/2.6.5/2.6.11/2.6.15/2.6.18/2.6.20, Mac OSX 10.4 server, NetBSD 3.1, OpenBSD 4.1, Solaris 10.1, Windows 2000/XP/Vista
Systems tested in the Internet:	AIX, AS/400, BSD/OS, Compaq Tru64, F5 Big-IP, HP-UX, IRIX, MacOS, NetApp NetCache, NetWare, OpenVMS, OS/2, SCO Unix, Solaris 8/9, SunOS 4, VM, Microsoft Windows NT4/98/Server 2003/2008
Servers tested in our lab.:	Abyss, Apache, Lighttpd, Microsoft IIS, Nginx
Servers tested in the Internet:	AOLserver, Araneida, Apache Tomcat, GFE, GWS-GFRE, IBM HTTP Server, Jetty, Jigsaw, LiteSpeed, Lotus-Domino, Mongrel, Netscape-Enterprise, OmniSecure, Oracle HTTP Server, Orion, Red Hat Secure, Redfoot, Roxen, Slinger, Stronghold, Sun Java System, thttpd, Twisted Web, Virtuoso, WebLogic, WebSiphon, Yaws, Zeus, Zope

6.1.2 Results for web servers in the Internet

In spite of the successful results above, OneProbe may still not be supported on some Internet paths because of middleboxes and customized TCP/IP stacks. We therefore extended the validation tests to websites in the Internet. We ran the Larbin web crawler [6] with slashdot.org as the starting URL (the same method used in [35]) to obtain 241,906 domain names and then randomly selected 38,069 websites from them. Based on the Netcraft database [29], the web servers came from 87 geographical locations, covering the 39 systems in Table 4 and 117 web server software. After excluding 195 of them that reset the TCP connections, we report the results from the remaining 37,874 websites below.

Successful (93.00%) These servers passed all tests.

Failures in the preparation phase (1.03%) These websites failed to return the expected $\{S1, S2\}$. Therefore, OneProbe could not start the probing phase.

Failures in test V0 (0.26%) Most websites in this set replied with $\{S3|4', S4|4'\}$, instead of the expected $\{S3|3', S4|4'\}$. That is, they sent response packets after receiving both probe packets.

Failures in test VR (5.71%) Some websites appeared

to have received an order-intact probe because of two kinds of response packets received from them: $\{S3|3', S4|4'\}$ and $\{S3|4', S4|3'\}$. Another set replied with $\{S3|3', S4|3'\}$; such behavior is similar to the problem of “failure to retain above sequence data” reported in [30]. The final set replied with $\{S3|2', S4|2'\}$, showing that they did not receive the reordered $C3'$, possibly due to packet drop by firewalls and intrusion detection systems. For example, Cisco IOS firewall drops reordered packets before release 12.4(11)T [13].

Since all the websites that failed test V1 also failed test VR, these failures are classified only under test VR.

6.2 Latency introduced by web servers

A common problem for non-cooperative measurement tools is that their delay measurement could be affected by the remote endpoint’s loading. In particular, a busy web server can introduce substantial latency during HTTP transaction processing [8].

6.2.1 Testbed and experiment setup

We setup a testbed to evaluate the impact of server-induced latency on the HTTP/OP measurement. The testbed consisted of a web server running Apache v2.2.3 and a probe sender where HTTP/OP and other measurement tools resided. Both machines were connected to each other through a router, which ran Click v1.6 [22] in kernel mode to emulate a fixed RTT of 25 milliseconds between them. Each machine, including the router, was equipped with a 1.7GHz Pentium 4 processor with 256MB memory running Linux v2.6.18 and connected to a 100Mbits/s LAN.

By adopting the approach described in [8], we set up two Surge web load generators [7] in separate machines that were directly connected to the web server. We experimented with a light load (20 Surge users from each generator) and a heavy load (260 Surge users from each generator). Each generator generated requests for objects selected from a set of 2000 distinct static files with size ranging from 78 bytes to 3.2MB. We conducted the same set of experiments for HTTP/OP and httping [19]. We included httping, because it is a common HTTP-based ping tool which uses HTTP HEAD and GET requests as probes to induce HTTP responses for RTT and round-trip loss measurement.

We restricted both HTTP/OP and httping to requesting five static text files of 20KB, 200KB, 2MB, 10MB, and 100MB available in the web server. We launched HTTP/OP using 30 TCP connections and periodic sampling with a rate of 20Hz (one probe every 50 milliseconds). All probe and response packets were 240 bytes in length. For httping, we used the default sampling rate of 1Hz and HEAD requests and responses for measurement. The httping’s probe and response packet sizes depended

on the URL specified in the HTTP request and the corresponding response.

For each load environment, we obtained the server-induced latency by measuring the difference between the arrival time of a probe packet at the server and the time of sending out the response packet that it has induced. Besides for HTTP/OP and httping, we measured the server-induced latency also for the initial HTTP request sent out in the HTTP/OP’s preparation phase. As discussed in section 4.1, this request is used for ramping up the server’s cwnd, therefore not used for measurement. We installed `tcpdump` at the server to capture all network traffic to and from the probe sender until we had obtained 150 latency samples for each experiment.

6.2.2 Server-induced latency

Figure 6 plots the cumulative distribution function (CDF) of the server-induced latency for HTTP/OP, httping, and HTTP/OP’s initial HTTP GET request under the light and heavy loads. The figure shows a significant latency occurred to both httping and the initial HTTP GET request. This start-up latency was reported for the Apache 1.3.0 architecture [8]. A similar delay of several milliseconds was also observed for a Google server to send out the first response packet for a request [12].

For the httping and initial HTTP GET request measurement, the server is required to invoke several expensive system calls (such as, `read()` and `stat()`) for processing the first request. Using the `strace` utility [4], we confirmed that the system calls invoked in the user space before sending out the response message was responsible for the start-up latency [2]. Besides, the start-up latency could last even longer because of additional backend server operations (e.g., the query delay of a Google search [12]).

HTTP/OP, on the other hand, avoids the substantial start-up latency, because it does not use the initial HTTP GET request for measurement. Moreover, when the first probe round starts, the response packets can be induced immediately after receiving a new TCP acknowledgment in a probe packet. Therefore, the overhead for the HTTP/OP measurement mainly comes from the data copying between the kernel space and devices. According to the `strace` results, the overhead of the data copy operations was low, because it was performed by invoking `sendfile()` to copy data from the file descriptor for the response message directly to a socket interface within the kernel.

Figure 6 also shows a much higher server-induced latency under heavy load for the httping and initial HTTP GET request measurement. The reason is that the server has less system resources for the start-up processing of httping’s HEAD request and the initial HTTP GET request. By avoiding the start-up latency, the HTTP/OP

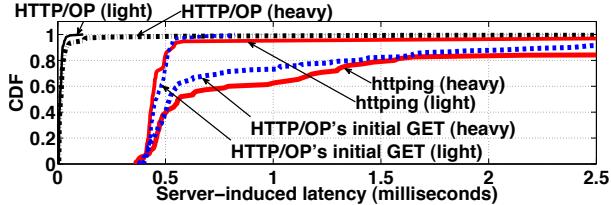


Figure 6: Server-induced latency experienced by HTTP/OP, httping, and HTTP/OP’s initial HTTP GET request under light and heavy loads.

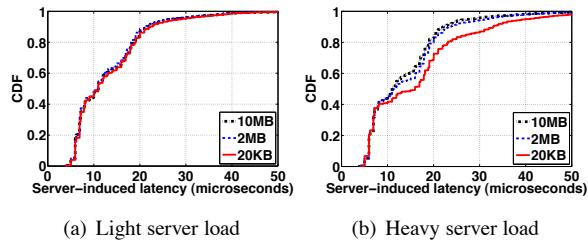


Figure 7: CDFs of the server-induced latency experienced by HTTP/OP.

measurement is also much less susceptible to the server load, as shown in Figure 6.

6.2.3 Effect of object size on server-induced latency

To evaluate the effect of the object size on the server-induced latency, we plot in Figure 7(a) (for light load) and Figure 7(b) (for heavy load) the CDFs of the server-induced latencies for the HTTP/OP measurement based on 4500 samples. For the sake of clarity, we show the results only for 20KB, 2MB, and 10MB. The observations obtained from them also hold for 200KB and 100MB.

Both figures show that the server-induced latency during the HTTP/OP measurement was very small: 80% of the samples were less than 30 microseconds. Therefore, the server-induced latency had negligible effect on the RTT measurement accuracy. Moreover, under a heavy server load, the latency was higher for a smaller object size, because HTTP/OP requested the server to load the requested objects more often. Under a light server load, however, the latency differences for the three object sizes were not significant. As a result, the server loading had more impact on the HTTP/OP measurement for small objects. Similar observations were reported in [8].

6.3 Resource consumptions of HTTP/OneProbe

6.3.1 System resources

Another important evaluation concerns the amount of system resources consumed by the HTTP/OP measurement in the probe sender and web server. We employed the same testbed but with different parameter settings. The web server hosted ten 61MB tarballs for retrieval. We ran HTTP/OP on the probe sender to randomly request the ten tarballs for 240 seconds using 1, 10, and 100

TCP connections and periodic sampling with five different rates: {1, 5, 10, 50, 100, 150}Hz. The probe and response packets had the same packet size of 1500 bytes.

We used `vmstat` [3] to measure the CPU and memory utilizations consumed by all Apache processes in the web server every second. At the same time, we measured the utilizations consumed by HTTP/OP in the probe sender. During the measurement, we ensured that no other routine processes were executed on both machines. Table 5 shows that the CPU utilizations were very low in all cases. Even when HTTP/OP used 100 concurrent TCP connections with a fine sampling rate of 150Hz, the average CPU utilizations of the probe sender and web server were still below 0.9% and 1.2%, respectively. The average memory utilizations (not shown here) of the probe sender and web server were also less than 2% and 6.3%, respectively, in all cases.

Table 5: The CPU utilizations consumed in the probe sender and web server during the HTTP/OP measurement.

Number of TCP connections	Sampling rates (Hz)	Average CPU utilizations (%)	
		Probe sender	Web server
1	1	<0.01	0.03
1	5	0.07	0.07
10	10	<0.01	0.27
10	50	0.07	0.70
100	100	0.17	0.77
100	150	0.87	1.17

We also performed similar experiments for three operating systems used by the web server: FreeBSD 6.2-RELEASE, Linux v2.6.18, and Microsoft Windows XP (SP2), and for three popular web server software with default settings: Lighttpd 1.4.18, Microsoft IIS 5.1, and Nginx 0.5.34. The CPU utilizations consumed by them during the HTTP/OP measurement ranged between 0.08% and 1.05%.

HTTP/OP incurs a small overhead to the probe sender, because it inspects only the TCP headers of the probe and response packets, and does not require saving the entire packet’s payload to the disk. Moreover, HTTP/OP applies `libpcap`’s packet filters to capture packets relevant to the path measurement and limits the amount of data captured from a packet.

6.3.2 Network I/O

To measure the network I/O for the HTTP/OP measurement, we conducted the measurement on the same testbed using five TCP connections and periodic sampling with a rate of 5Hz. HTTP/OP requested files of 2MB, 10MB, and 100MB for 240 seconds. The probe and response packet sizes were 1500 bytes. We used the `sar` utility [5] to measure the network I/O from the web server side in terms of the number of packets per second (`pkts/s`) and bytes per second.

The results in Table 6 are very close to the expected results of 10 pkts/s ($5\text{Hz} \times 2$ packets) and 15000 bytes/s (10 pkts/s $\times 1500$ bytes/pkt) for both reception (Rcv) and transmission (Tmt). The results are slightly higher than the expected results, because of the additional packets for the TCP connection establishment and termination. Table 6 also shows that the network I/O stays almost the same for different object sizes, because it depends only on the probe and response packet sizes.

Table 6: Network I/O for the HTTP/OP measurement.

Object sizes (MB)	Rcv (pkts/s)	Tmt (pkts/s)	Rcv (bytes/s)	Tmt (bytes/s)
2	11.36	11.52	15598	16508
10	11.35	11.52	15598	16511
100	11.34	11.48	15590	16485

7 Measurement experiences

This section reports our recent experience of deploying HTTP/OP for Internet path measurement. All the measurement results reported here were obtained from an HTTP/OP deployment at a Hong Kong data center. The full set of results and the measurement setup are available from [11].

7.1 Diurnal RTT and loss patterns

This set of measurement results was obtained from a set of web servers hosting the last Summer Olympic Games. HTTP/OP sent a probe every 500 milliseconds, continuously for one minute, and the same probing pattern repeated after idling for four minutes. The entire measurement was smoothly conducted for over a month.

Figure 8 shows the RTT and round-trip loss rate measurement for one of the paths. The HTTP/OP measurement captured clear diurnal RTT and round-trip loss patterns. The peak loss rates also coincided with the daily high RTT periods. A positive correlation between RTT and loss rate was also reported by observing packet losses at bottleneck queues in a ns-2 simulation study [9]. For temporal correlation, the high RTT periods were longer and the intensity of the peak loss rates were higher on weekends.

Studying the correlation of RTT and packet loss rate is important for predicting network congestion from end hosts [9]. HTTP/OP provides a more accurate measurement of their correlation, because it can sample an Internet path with more fine-grained and uniform sampling, and over a long duration. HTTP/OP’s intrusion to the path is also minimal, thus minimizing the self-induced bias. For the purpose of comparison, the measurement in [27] was conducted for five days and for each day each run was executed every two hours, and it introduced between 6 and 20 MB in each run.

7.2 Discrepancy between Ping and OneProbe RTTs

This set of results is also part of the Olympic Games measurement. Besides HTTP/OP, we also deployed ICMP Ping and other tools for path measurement. To compare their results accurately, the tools were configured to measure the same path at the same time. Figure 9 shows the RTT measurement obtained by HTTP/OP and Ping for one of the paths. The figure shows that for the first few days their RTTs consistently differed by around 100 milliseconds on the peaks, but they were similar on the valleys. As a result, the Ping measurement underestimated the actual RTT experienced by TCP data packets by as much as 70%! Moreover, due to an (possibly network configuration) event unseen to Traceroute, their RTTs “converged” at 12 Aug. 2008 16:39 UTC. At the same time, the forward-path loss rate dropped significantly after this convergence point. Therefore, non-data probes may not measure the actual path quality experienced by data packets.

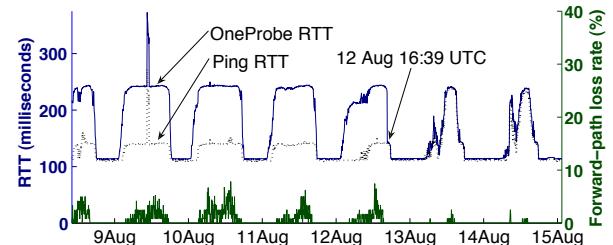


Figure 9: Discrepancy in the RTT measurement obtained by HTTP/OP and Ping for a Summer Olympics web server.

7.3 Asymmetric loss rates and loss-pair RTTs

This set of results is also part of the Olympic Games measurement. For all the paths in this set of measurement, the reverse-path losses dominated the round-trip loss rates, and in some cases the packet losses occurred only on the reverse paths. These results are consistent with web’s highly asymmetric traffic profile. Moreover, we conducted a parallel measurement to the same servers but with different reverse paths, but we did not observe packet losses from this set of measurement. Therefore, the packet losses were believed to occur on the reverse paths close to the web servers but not in the web servers.

Moreover, HTTP/OP can measure the loss-pair RTT. A probe packet-pair or a response packet-pair is considered a *loss pair* if *only* one packet is lost to the pair [23]. Loss-pair analysis has been shown useful in estimating bottleneck buffer sizes of droptail routers and characterizing packet dropping behavior [23]. However, in the absence of a suitable measurement tool, the loss-pair analysis has so far been analyzed using simulations and restricted to round-trip loss pairs.

Figure 10 shows the forward-path and reverse-path

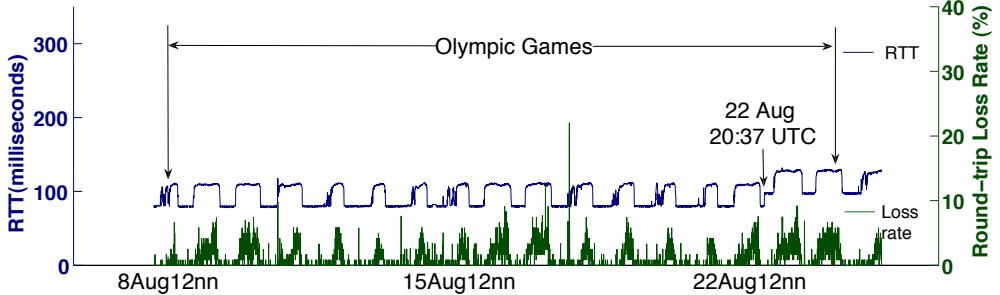


Figure 8: Time series of RTT and round-trip loss rates obtained by HTTP/OP for a Summer Olympics web server.

loss-pair RTTs for one of the paths, and the RTT was measured for the first packet in the pair (and the second was lost). The loss-pair RTTs are superimposed with the corresponding RTT time series to identify which parts of the RTT time series the loss pairs were located. The figure shows that almost all the loss-pair RTTs on the forward path were clustered on the RTT peaks, suggesting that the packets were dropped in a drop-tail router on the forward path. However, the reverse-path loss-pair RTTs behaved very differently. While many loss pairs saw the highest RTT, there were also many others seeing other RTT values, including the lowest RTT. Therefore, the packet dropping behavior is more similar to that exhibited by a random-early-drop router.

7.4 Effect of packet size on reordering rates

This set of measurement results was obtained from a PlanetLab node [32]. The HTTP/OP measurement revealed that this path experienced persistent, high reordering rates on both forward and reverse paths over one week. We experimented with three combinations of packet sizes: {280, 280}, {280, 1420}, and {1420, 280}, where the first is the probe packet size in bytes and the second response packet size in bytes. Note that the current non-cooperative tools cannot measure the reverse-path reordering rate for different packet sizes.

Figure 11(a) depicts how the packet size affected the reordering rate for the forward path. The reordering pattern for {280, 280}, which is not included in the figure, is similar to that for {280, 1420}. A comparison of the three results therefore concludes that a smaller probe packet is more prone to packet reordering. This finding is consistent with the results obtained from a cooperative measurement study [16] and TBIT measurement [28].

Figure 11(b) shows the distinctive reordering rates on the reverse path for the three packet size combinations. Same as the forward-path reordering, a smaller response packet size is more prone to packet reordering. Thus, the case of {280, 1420} suffered from the least reordering. Surprisingly though, the reordering rate for {280, 280} was distinctively higher than that of {1420, 280}, although they had the same response packet size. A pos-

sible explanation is that smaller probe packets will reach the server with a smaller inter-packet interval. They will therefore induce two response packets also with a smaller interval, and the occurrence of packet reordering generally increases with a shorter inter-packet interval.

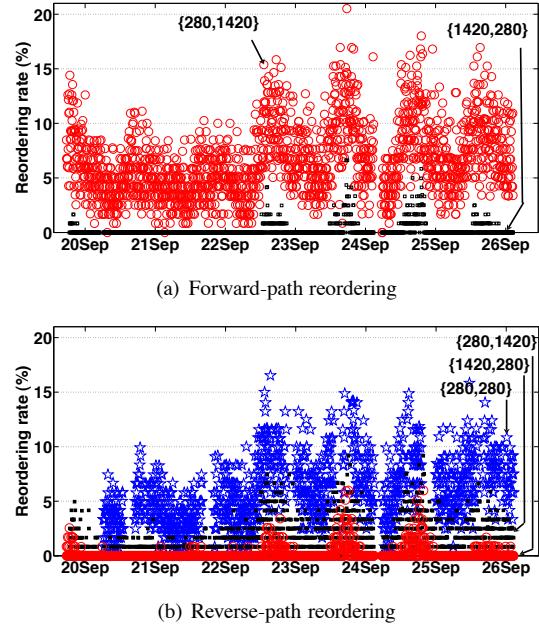


Figure 11: Time series of forward-path and reverse-path packet reordering rates obtained by HTTP/OP for a PlanetLab node.

8 Conclusions

In this paper, we presented OneProbe, a new TCP probing method, and HTTP/OneProbe, an implementation of OneProbe for HTTP/1.1 to induce sufficient HTTP data for continuous measurement. HTTP/OneProbe's path measurement is reliable, because the probes and induced response packets are legitimate HTTP/TCP data packets, and the probes are based on TCP's basic fundamental transmission mechanisms. OneProbe can also sample RTT, packet loss rates on the forward and reverse paths, and packet reordering rates on the forward and reverse paths at the same time using the same probe. We per-

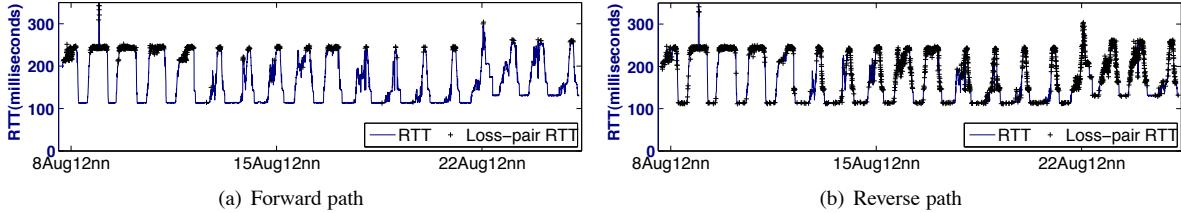


Figure 10: Time series for the loss-pair RTTs obtained by HTTP/OP for a Summer Olympics web server.

formed extensive experiments to validate the correctness of the probe responses, to evaluate the performance and accuracy of HTTP/OneProbe, and to monitor network paths for over a month. We are currently introducing new path metrics, such as capacity and available bandwidth, to OneProbe.

Acknowledgments

We thank the five anonymous reviewers for their critical reviews and suggestions and Mike Freedman, in particular, for shepherding our paper. We also thank Waiting Fok for preparing the colorful Internet measurement plots. This work is partially supported by a grant (ref. no. ITS/152/08) from the Innovation Technology Fund in Hong Kong.

References

- [1] Apache: HTTP server project. <http://httpd.apache.org/>.
- [2] Apache Performance Tuning. <http://httpd.apache.org/docs/2.2/misc/perf-tuning.html>.
- [3] procps. <http://procps.sourceforge.net/>.
- [4] strace. <http://sourceforge.net/projects/strace>.
- [5] SYSSTAT. <http://pagesperso-orange.fr/sebastien.godard/features.html>.
- [6] S. Ailleret. Larbin: Multi-purpose web crawler. <http://larbin.sourceforge.net/>.
- [7] P. Barford and M. Crovella. Generating representative workloads for network and server performance evaluation. In *Proc. ACM SIGMETRICS*, 1998.
- [8] P. Barford and M. Crovella. Critical path analysis of TCP transactions. *IEEE/ACM Trans. Networking*, 9(3), 2001.
- [9] S. Bhandarkar, A. Reddy, Y. Zhang, and D. Loguinov. Emulating AQM from end hosts. In *Proc. ACM SIGCOMM*, 2007.
- [10] J. Bolot. End-to-end packet delay and loss behavior in the Internet. In *Proc. ACM SIGCOMM*, 1993.
- [11] R. Chang, E. Chan, W. Fok, and X. Luo. Sampling TCP data-path quality with TCP data probes. In *Proc. PFLDNet*, 2009.
- [12] Y. Cheng, U. Holzle, N. Cardwell, S. Savage, and G. Voelker. Monkey see, monkey do: A tool for TCP tracing and replaying. In *Proc. USENIX Annual Technical Conference*, 2004.
- [13] Cisco Systems. TCP out-of-order packet support for Cisco IOS firewall and Cisco IOS IPS. <http://www.cisco.com/>, 2006.
- [14] E. Corell, P. Saxholm, and D. Veitch. A user friendly TSC clock. In *Proc. PAM*, 2006.
- [15] S. Floyd and E. Kohler. Tools for the evaluation of simulation and testbed scenarios. Internet-draft draft-irtf-tmrg-tools-05, February 2008.
- [16] L. Gharai, C. Perkins, and T. Lehman. Packet reordering, high speed networks and transport protocol performance. In *Proc. IEEE ICCCN*, 2004.
- [17] GNU Wget. <http://www.gnu.org/software/wget/>.
- [18] A. Haeberlen, M. Dischinger, K. Gummadi, and S. Saroiu. Monarch: A tool to emulate transport protocol flows over the Internet at large. In *Proc. ACM/USENIX IMC*, 2006.
- [19] F. Heusden. htpting. <http://www.vanheusden.com/htpting/>.
- [20] J. Postel (editor). Transmission control protocol. RFC 793, IETF, September 1981.
- [21] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, IETF, May 1992.
- [22] E. Kohler. The Click Modular Router Project. <http://read.cs.ucla.edu/click/>.
- [23] J. Liu and M. Crovella. Using loss pairs to discover network properties. In *Proc. ACM IMW*, 2001.
- [24] M. Luckie, Y. Hyun, and B. Huffaker. Traceroute probe method and forward IP path inference. In *Proc. ACM/USENIX IMC*, 2008.
- [25] X. Luo and R. Chang. Novel approaches to end-to-end packet reordering measurement. In *Proc. ACM/USENIX IMC*, 2005.
- [26] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *Proc. ACM SOSP*, 2003.
- [27] J. Martin, A. Nilsson, and I. Rhee. Delay-based congestion avoidance for TCP. *IEEE/ACM Trans. Networking*, 11(3), 2003.
- [28] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the Internet. *ACM CCR*, April 2005.
- [29] Netcraft Services. <http://uptime.netcraft.com/up/accuracy.html>.
- [30] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP implementation problems. RFC 2525, IETF, March 1999.
- [31] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. RFC 2330, IETF, May 1998.
- [32] PlanetLab. <http://www.planet-lab.org/>.
- [33] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999.
- [34] S. Savage. Sting: a TCP-based network measurement tool. In *Proc. USENIX Symp. Internet Tech. and Sys.*, 1999.
- [35] R. Sherwood and N. Spring. A platform for unobtrusive measurements on PlanetLab. In *Proc. USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, 2006.
- [36] R. Sherwood and N. Spring. Touring the Internet in a TCP sidecar. In *Proc. ACM/USENIX IMC*, 2006.
- [37] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public Internet measurement facility. In *Proc. USENIX Symp. Internet Tech. and Sys.*, 2003.
- [38] L. Wenwei, Z. Dafang, Y. Jinmin, and X. Gaogang. On evaluating the differences of TCP and ICMP in network measurement. *Computer Communications*, January 2007.

StrobeLight: Lightweight Availability Mapping and Anomaly Detection

James W. Mickens, John R. Douceur, William J. Bolosky
Microsoft Research
mickens,johndo,bolosky@microsoft.com

Brian D. Noble
University of Michigan
bnoole@umich.edu

Abstract

Large-scale distributed systems span thousands of independent hosts which come online and go offline at their users' whim. Such availability flux is ostensibly a key concern when systems are designed, but this flux is rarely measured in a rich way post-deployment, either by the distributed system itself or by a standalone piece of infrastructure. In this paper we introduce StrobeLight, a tool for monitoring per-host availability trends in enterprise settings. Every 30 seconds, StrobeLight probes Microsoft's entire corporate network, archiving the ping results for use by other networked services. We describe two such services, one offline and the other online. The first service uses longitudinal data collected by our StrobeLight deployment to analyze large-scale trends in our wired and wireless networks. The second service draws live StrobeLight measurements to detect network anomalies like IP hijacking in real time. StrobeLight is easy to deploy, requiring neither modification to end hosts nor changes to the core routing infrastructure. Furthermore, it requires minimal network and CPU resources to probe our network of over 200,000 hosts.

1 Introduction

As distributed systems are built at increasingly larger scales, it becomes more difficult to understand the relationship between host availability and distributed system performance. Loosely coordinated, independently administered hosts display a wide variety of availability patterns [8, 10, 25]. Providing robust services atop this churning substrate requires substantial effort during the design and implementation of the distributed system. Thus, all distributed systems are guided by at least a crude characterization of host availability in the deployment environment.

Unfortunately, once these systems are deployed, they rarely include a component for collecting and analyzing system-wide, fine-grained availability data. Historical availability traces exist (e.g., [8, 10]), but they were collected by one-shot tools that were not intended to be permanent, stable pieces of the distributed infrastructure.

The permanent monitoring tools in existence often focus on monitoring path characteristics, not individual host availability, so they issue measurements to and from a small set of vantage points. For example, RON [4] and iPlane [24] track latency and loss rates between a set of topologically diverse end points, but these machines are assumed to be highly available and small in number; no mechanism is provided for testing individual host availability inside a stub network. CoMon [29] provides uptime monitoring for individual PlanetLab hosts, but it does not scale to hundreds of thousands of machines. Furthermore, it requires modifications to end hosts, which may be difficult in non-academic settings where people are leery of installing new software.

In overlays like Pastry [32] and storage systems like TotalRecall [9], hosts probe the availability of select peers, but this data is not archived in a public directory, preventing global analysis. Schemes to distribute such data exist [20, 26], but large-scale data mining is difficult due to the number of wide-area data fetches required, as well as the need to perform cryptographic calculations to verify measurements submitted by untrusted peers.

The lack of a persistent infrastructure for availability monitoring is unfortunate because it could benefit a wide variety of systems. For example, distributed job allocators [5] could use historical uptime data in concert with availability prediction [25] to assign high priority tasks to machines that are likely to be online for the expected duration of the job. Distributed storage systems could also use a live feed of availability measurements to guide object placement and increase data availability [1].

To address such needs, we introduce StrobeLight, a tool for measuring availability in an enterprise setting containing hundreds of thousands of hosts. StrobeLight issues active probing sweeps at 30 second intervals, archiving ping results for the benefit of other distributed services that might find them useful. We describe two examples of such services. The first is an offline data-miner for longitudinal availability traces; such an application might be useful for distributed storage systems trying to make decisions about replica allocation. The second StrobeLight service monitors network-wide availability in real time,

raising alarms for anomalies such as network partitions and IP hijacks. StrobeLight detects such problems using a new abstraction called an *availability fingerprint*. Under normal conditions, a subnet’s fingerprint changes very slowly. Thus, StrobeLight raises an alert when the similarity between consecutive fingerprints falls below a threshold. Using Planetlab experiments, simulations, and a real enterprise deployment, we show that our detection system is accurate and fast.

By using standard ICMP probes to test availability, StrobeLight avoids the need to install new software on end hosts or deploy new infrastructure within the routing core. By collecting data from a few centrally controlled vantage points, StrobeLight avoids the trust and complexity issues involved with distributed solutions while making it easy for other systems to access availability data.

This paper provides three primary contributions. From the technical perspective, it demonstrates that frequent, active probing of a large host set is cheap and practical. From an analytical perspective, it introduces new techniques for analyzing availability traces that contain temporal gaps (see Section 3.3). Finally, the paper introduces a new, fine-grained trace of wired and wireless availability in a large corporate environment. Using this trace, we can validate results from previous studies that used coarser-grained data [10, 25]. We also discover an interesting property about the stability of subnet availability. From the qualitative perspective, subnet uptime is consistent across weeks—for example, the relative proportion of diurnal hosts is unlikely to change. However, from the quantitative perspective, subnet availability may fluctuate by more than 25% across a month (see Section 3.4.2).

2 Design and Implementation

The design of our availability measurement system was guided by three principles. First, keep the system simple. Second, make the system unobtrusive. Third, collect fine-grained data.

Keep it simple. Our primary design principle was to keep everything simple, a philosophy reflected in many different ways. We wanted to avoid solutions which required new software to be installed on end hosts, an arduous task that is difficult to justify on a corporate-wide basis. Similarly, we hoped to avoid major modifications to our internal routing infrastructure. Large-scale decentralization of the probing infrastructure was not a primary concern. Although coordinated distributed monitoring has certain benefits, previous experience had taught us that the road to a bug-free distributed protocol is fraught with peril [11]. Thus, we thought hard about the costs and benefits of a coordinated peer-to-peer design, and ultimately rejected it. One motivating factor was our development of analysis techniques which tolerate temporal

gaps in availability data (see Section 3.3). These techniques shifted the payoff curve between the better coverage and robustness of a distributed, coordinated solution and the reduced complexity of a centralized one.

Don’t annoy the natives. We wanted a system that was unobtrusive—we did not want our measurement activity to disrupt normal network traffic or add significant load. We also required a straightforward mechanism to turn off measurement activity in specific parts of the network. The latter was important because previous experience had taught us that at some point, our new network infrastructure would break someone else’s experiment or interact with other components in unexpected ways. When such scenarios arose, we wanted the capability to quickly remove the friction point.

Collect high-resolution data. We wanted our tool to collect per-host availability statistics at a fine temporal granularity. This would allow us to validate previous empirical studies which used coarser data sets [10, 25]. It would also make the service more useful for anomaly detection, since disruptions like IP hijacking may only last for a few minutes [31].

These design considerations led to several “non-goals” for our system.

Infinite scalability is overkill. Our solution only needed to scale to the size of an enterprise network containing hundreds of thousands of hosts. Building a measurement system to cover an arbitrary number of hosts in an arbitrary number of administrative domains would have been extremely challenging. For example, active availability probing from foreign domains might trigger intrusion detection systems. Organizations might also be reluctant to provide outside access to DNS servers and other infrastructure useful for identifying “live” end hosts.

Complete address disambiguation is difficult. Another barrier to performing arbitrary-scale, cross-domain host monitoring is the widespread use of NATs, firewalls, and DHCP. These technologies can create arbitrary bindings between hosts and IP addresses, and prevent some machines from being seen by external parties. Devising a comprehensive monitoring system that can pierce this heterogeneous cloud of addressing policies is an important research topic. However, this goal was beyond the scope of our project. By focusing on enterprise-level solutions, we hoped to avoid many of the issues mentioned above; NATs were relatively rare in our corporate environment, and we could configure our firewalls to trust packets generated by our new monitoring system.

2.1 The Winning Design: StrobeLight

As shown in Figure 1, we eventually chose a centralized architecture in which a single server measured availability throughout our entire network. To determine which IP addresses to test, the server would download hostname/IP

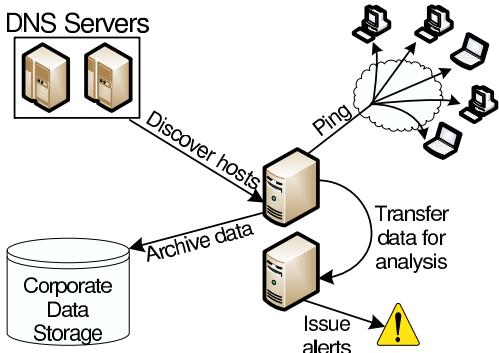


Figure 1: StrobeLight Architecture

mappings from corporate DNS servers. It would then test host availability using standard ping probes issued at intervals of 30 seconds. Recent probe results would be transferred to an analysis server for real-time anomaly detection, and longitudinal data would be archived in the corporation’s standard distributed data store.

This design, which we named StrobeLight, was very attractive from the implementation and deployment perspectives. No new code would have to be pushed to end hosts or internal routers, and the only additional hardware required would be the probing server and the analysis engine. We also expected the probing process to have a light footprint. The total volume of request/response traffic would be trivial compared to the overall traffic level in the corporate network. Furthermore, we would not have to deal with control or synchronization issues that might arise in a more decentralized design. Our main concerns involved performance and fault tolerance. We feared that a single server might be overloaded by sending probes for hundreds of thousands of machines every 30 seconds. A centralized probing design also had obvious ramifications for fault robustness. Despite these weaknesses, we committed to the single-server design due to its relative ease of implementation, and we pledged to revisit the design if we encountered undue difficulties after deployment.

2.2 Implementation and Deployment

The core probing infrastructure was deployed first. The pinging daemon, consisting of 2,200 lines of C++ code, runs on a standard desktop PC with a 3.2 GHz CPU, 2 GB of RAM, and a gigabit Ethernet card; this machine resides within a corporate subnet in Redmond, WA. At boot time, the daemon reads an exclusion file which specifies the set of IP prefixes that should never be pinged. This file allows us to selectively exclude parts of the network from our probing sweeps. To determine which IP addresses to ping, the daemon downloads zone files from Microsoft’s DNS servers at 2:10 AM each day. At any given moment, these zone files contain entries for over 150,000 IP

addresses scattered throughout the world. This set of addresses evolves over time due to the introduction of new hosts and the decommissioning of old ones.

Due to these factors, an address may not appear in every DNS snapshot. Since StrobeLight only probes the addresses mentioned in the zone data, an IP may have gaps in its availability history. To deal with these gaps, StrobeLight describes the availability of an address as online, offline, or unknown. The first two categories result from the outcome of a ping probe, whereas the third is assigned to an IP which was not probed at a particular time.

Once the probing daemon had produced a sizable archive of availability data, we were able to test the offline analysis engine. This engine, totaling about 5,000 lines of C++ code, provides a set of low-level classes to represent per-host availability. It also defines a high-level query interface for use by data mining programs. We used this interface to generate the results in Section 3. Importantly, the interface defines a subnet of size N as a set of N consecutive and *allocated* IP addresses; the querier chooses the starting address, N , and the time period over which “allocated” is defined. An address is considered allocated during a given time period if it appeared in a zone file at least once during that period. In practice, we often set N to a small number like 256 and investigate the subnets contained within a Class A or B prefix.

2.3 Operational Experiences

The probing server has run with few interruptions for almost three years, and it has not struggled with the network load generated by the ping sweeps. We currently spread each sweep across 25 seconds to avoid load spikes on our shared network infrastructure, but brief “full throttle” experiments show that our current prober can scan 270,000 hosts in 7.9 seconds (roughly 35,000 hosts a second).

In general, our ping traffic has not bothered the other members of our network. We occasionally receive emails from the network support staff when they unveil a new intrusion detection system and they conclude that our probing machine is infected with an IP-scanning virus; these incidents became rarer after we explained that StrobeLight was a piece of permanent infrastructure. We also received a complaint from another research group who claimed that our pings were causing problems for their wireless devices. After generating the appropriate exclusion file and restarting the daemon, we received no more complaints.

3 Application 1: Offline Analytics

In this section, we describe one application of StrobeLight, using it to gather long-term availability data for offline analysis. Such data could be used in several

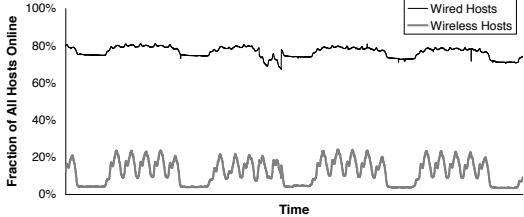


Figure 2: Global availability (10/21/2005 to 11/21/2005)

ways, e.g., to guide replication policy in a distributed data store [1, 9, 25]. In this section, we use the data in a more exploratory fashion, looking for interesting patterns in our wired and wireless networks. We restrict our analysis to IP addresses which appeared in at least 95% of the daily DNS snapshots. During the time period examined below, this included 138,801 wired IPs and 11,670 wireless IPs. In our corporate environment, the DHCP lease time is 20 days for wired machines and 3 hours for wireless ones. Thus, a wireless address is likely to be bound to multiple machines over the course of the day. Although we often refer to “hosts” and “IP addresses” interchangeably, the true unit of uniqueness is an address, not a host.

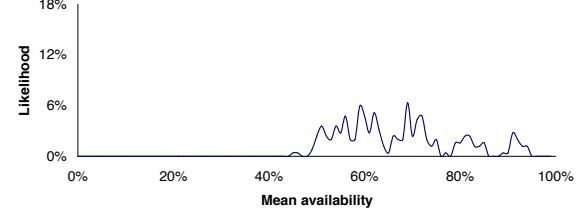
3.1 Global Trends

Figure 2 depicts aggregate availability fluctuations from October 21 to November 21 of 2005. The bulk of Microsoft’s machines reside in the American west coast, so both the wired and wireless networks show large-scale diurnal trends aligned with the work day in this time zone. However, during these large-scale surges and declines in availability, there are regular, smaller-scale peaks and valleys. These additional periodic cycles are driven by phase-shifted diurnal behavior amongst Microsoft hosts in Europe and the Middle East.

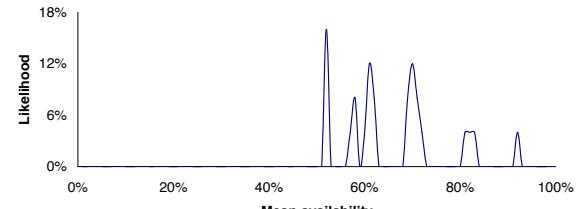
Comparing the two curves in Figure 2, we see that wireless IP addresses are much less likely to be associated with online hosts. However, the wireless network demonstrates stronger diurnal trends than the wired network. We investigate this issue further in Section 3.3.

3.2 Subnet-level Trends

We define the mean availability of a subnet as its average fraction of online hosts. Figure 3 shows the distribution of mean subnet availability in the wired network for subnets of size 256 and 2048. In both cases, mean subnet availability is always higher than 40%. Increasing the subnet size causes probability mass to coalesce around several regions of mean availability. This is a discretization artifact, since increasing the subnet size without increasing the total number of hosts results in fewer subnets to examine and less smoothness in the resultant distribution.



(a) 256 hosts per subnet



(b) 2048 hosts per subnet

Figure 3: PDF for mean subnet availability (wired)

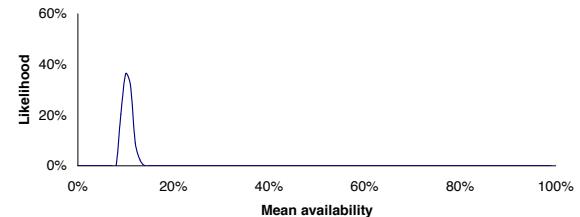
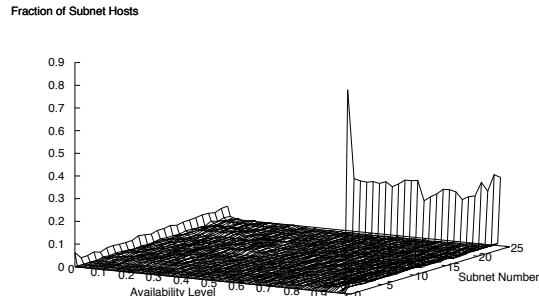


Figure 4: PDF for mean subnet availability (wireless)

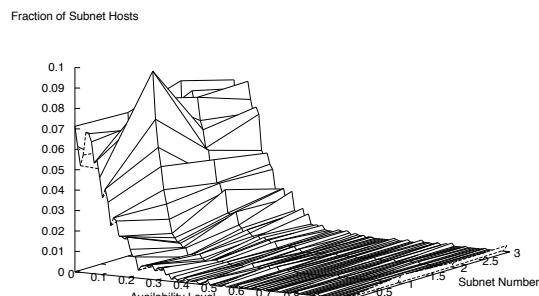
As expected, Figure 4 shows that wireless subnets have much lower mean availability. Figure 4 shows results for a subnet size of 256, but increasing the subnet size to 2048 results in an almost identical availability distribution. The relative lack of discretization artifacts is due to the greater homogeneity of wireless host availability. Figure 5 shows the distribution of per-host uptime fractions within each subnet. Each wired subnet has a skewed bimodal distribution, with a plurality of hosts having very high uptime and a smaller fraction having very low uptime. However, in every wired subnet, roughly 50% of the probability mass is spread across the “plateau” between the two modes. In contrast, the wireless subnets look more unimodal, with the majority of hosts having very low availability and much less probability mass sheared away from the mode.

3.3 The Availability of Individual Hosts

To understand the lower-level dynamics driving aggregate availability, we modified our previous taxonomy for classifying the uptime behavior of individual hosts [25]. In the unmodified scheme, a host is declared *always-on* if its uptime is greater than 90% and *always-off* if its uptime is less than 10%. If a host fails these tests, its availability signal is converted into the frequency domain using



(a) Wired subnets (2048 hosts per subnet)



(b) Wireless subnets (2048 hosts per subnet)

Each curve on the “availability level” axis is a pdf for per-host uptime fractions in a particular subnet. In each figure, the pdfs are sorted by standard deviation, with higher subnet numbers indicating larger standard deviations. The trends depicted in each graph are insensitive to subnet size.

Figure 5: Per-host availability within a subnet

a Fourier transform. If the resultant profile demonstrates harmonic peaks in the daily and weekly spectra, the host is labeled *diurnal*. If the spectral curve resembles the curve $1/f$, i.e., it contains large amounts of low frequency energy, the host is labeled as *long stretch*, meaning that it has long, uninterrupted periods of uptime and downtime. Nodes failing all four tests are labeled as *unstable*. Such a designation usually implies that the host’s availability is difficult to predict.

The standard algorithms for Fourier decomposition assume that signals are sampled at a uniform rate and that no samples are missing. In our data set, the assumption of a uniform sampling rate was almost always true, since the vast majority of probe sweeps were separated by 30 second intervals. However, missing samples were fairly common for two reasons. First, our network used DHCP

to assign IP addresses to physical machines. When an address was dormant (i.e., unassigned), it did not show up in our zone files, meaning that we did not collect availability data for it during the dormant period. Second, the DNS servers occasionally failed, or misbehaved and returned extremely small zone files. Both of these phenomena introduce brief probing gaps for many hosts.

To deal with missing samples, we replaced the Fourier analyses with two entropy-based techniques. To determine whether an availability signal contained diurnal patterns, we adapted Cincotta’s method for period detection in irregularly sampled time series [14]. Let $a_t \in \{0, 1\}$ be the value of an availability signal at time t . Given a hypothetical period τ , we calculate the phase of each a_t as $\phi_t = \frac{t}{\tau} - \text{nearestInteger}(\frac{t}{\tau})$; note that $\phi_t \in [-0.5, 0.5]$. We can interpret each (ϕ_t, a_t) pair as a coordinate in $\phi \times a$ space. If the hypothesized period τ is close to the signal’s actual period (or a harmonic of it), the (ϕ_t, a_t) points will cluster in the coordinate space. This means that if we divide the coordinate space into bins, the resultant bin distribution will have low entropy. If the hypothesized period is not the signal’s true period, points will be scattered throughout the $\phi_t \times a_t$ space and the bin distribution will have high entropy.

To determine whether an availability signal contains diurnal patterns, we check whether the entropy for a τ of 24 hours is less than the entropy for a τ of 23 hours. Availability signals with complex diurnal patterns may have entropy dips in other places, but finding one for a τ of 24 is sufficient for our purposes.

To determine whether an availability signal contains long-stretch behavior, we use an approximate entropy test [30]. Suppose that we have an arbitrary window of k consecutive samples from the signal. We define $\text{ApEn}(k)$ as the additional information conveyed by the last sample in the window, given that we already know that previous $k - 1$ samples. Low values of $\text{ApEn}(k)$ indicate regularity in the underlying signal. In particular, if we know that a host is not always-on, always-off, or diurnal, but it still has a low $\text{ApEn}(k)$, it is likely that the uptime regularity is driven by long-stretch behavior.

The choice of window size k is driven by the time scale over which “long stretch” is defined; k should be small enough that a stretch contains several windows, but not so small that $\text{ApEn}(k)$ measures the incidence of small k -grams that are actually pieces of larger, more complex availability patterns. In the results presented below, we used a k of 8 and sampled our availability trace in steps of 15 minutes. This meant that we looked for long-stretch behavior at a time scale of roughly two hours. We defined hosts as long-stretch if their availability signal had an $\text{ApEn}(8)$ of less than 0.16. This cutoff was determined by hand, but our results were not very sensitive to the exact value.

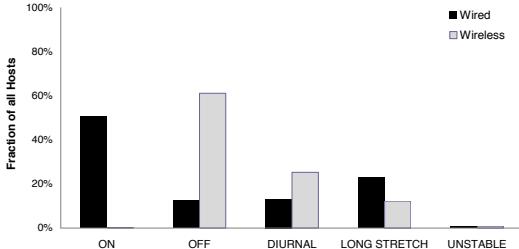


Figure 6: Availability taxonomy

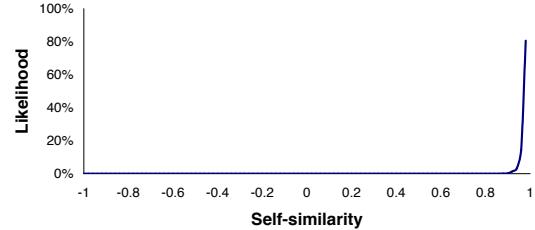
Figure 6 depicts the availability taxonomy for the wired and wireless networks. We found that roughly half of the wired hosts were always online. This result is congruent with smaller-scale observations of the Microsoft network which used an hourly sampling period [10, 25]. Indeed, the fact that the always-on fraction is the same at a finer sampling granularity implies that the natural time scale for availability fluctuation in wired corporate environments is hours, not minutes. This claim is further validated by the fact that almost none of the wired hosts had unstable availability. In other words, if a host was not always-on, always-off, or diurnal, then it at least had availability that was stable across one or two hours.

The wireless network was dominated by always-off machines, which comprised 61% of all hosts. The wireless network had almost twice as many diurnal machines as the wired network (25% versus 13% respectively) but almost half as many long-stretch hosts (12% versus 23%). These trends were unsurprising. In contrast to desktop machines that were always “plugged in,” wireless devices with limited battery lives were more likely to have shorter sessions. Also, users often took these devices home at the end of the day, removing them from the physical proximity of a corporate access point. Thus, wireless connectivity exhibited stronger diurnal patterns and less long-stretch behavior than wired uptime.

3.4 Availability Fingerprints

Up to this point, we have investigated aggregate availability trends over a five week window. However, many network anomalies occur over a much smaller time scale. For example, an IP hijacking attack might only last for several minutes [31], and BGP misconfigurations can be just as transient [13].

Both types of anomaly change the mapping between IP addresses and physical hosts. In a hijacking attack, an entire range of IPs is bound to a different set of physical machines; similarly, a misconfigured router can cause arbitrary desynchronizations. Active availability probing can detect such problems if three conditions are true. First, the probing interval must be less than the duration of the desynchronization episode, lest the anomaly escape undetected between probing sweeps. Second, in the absence



Subnet self-similarity between successive probing sweeps is very high. The graph depicts results for wired subnets of size 256, but the outcome is insensitive to subnet size. The results are extremely similar for wireless subnets.

Figure 7: PDF for self-similarity of delta fingerprints (15 minute probe interval)

of anomalies, a subnet’s availability “fingerprint” must be stable across multiple consecutive probing periods. This gives us confidence that when the fingerprint changes, an actual problem has arisen. Third, at any given moment, the availability fingerprint for each subnet should be globally unique. This allows us to detect routing problems in which two subnets have their IP bindings swapped.

With these desired characteristics in mind, we can provide a formal definition of a fingerprinting system. Given a specific subnet and a time window of interest, a fingerprinting algorithm examines per-host availability trends during that window and produces a bit-string that is a function of those trends. A fingerprinting system also defines a distance metric which determines the similarity of two fingerprints. To detect an anomaly in a subnet, we maintain a time series of its fingerprints and raise an alarm if the most recent fingerprint is too dissimilar from the previous one.

In the remainder of this section, we provide a concrete description of a fingerprinting system and evaluate its performance on trace data collected by StrobeLight. We focus on basic issues such as how a subnet’s fingerprint evolves over time, and the accuracy with which we can distinguish two subnets based solely on their fingerprints. We present more applied results in Section 4, where we show how fingerprints can be used to detect anomalies within the enterprise and across the wide area.

3.4.1 Delta Fingerprints

During a single probe sweep, we test the availability of each known host in our network. Given a subnet of size s , we represent its probe results as an s -bit vector where a particular bit is 1 if the corresponding host was online and 0 if the host was offline or unknown (remember that a host is not probed if it is not mentioned in the current DNS mapping). We call such a vector an instantaneous or delta fingerprint because it represents a snapshot of subnet availability at a specific time.

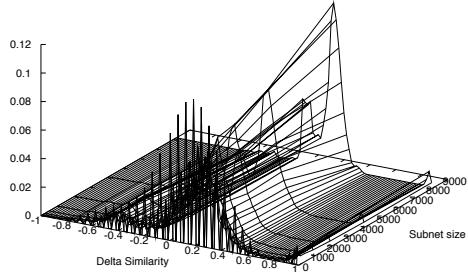


Figure 8: PDF for instantaneous cross-subnet similarity (15 minute probe interval)

A natural distance metric for two delta fingerprints is the number of bit positions with equivalent values. Thus, we define the similarity of two fingerprints as the number of equivalent bit positions divided by s and normalized to the range $[-1, 1]$. For example, if two fingerprints match in half of their bit positions, they will have a similarity of 0. If they match in all positions or no positions, they will have a similarity of 1 or -1 respectively.

Given the availability probing period ρ , we define a subnet’s self-similarity as the expected similarity of its fingerprints at time t and time $t + \rho$. Figure 7 depicts the pdf for self-similarity in the wired network with a ρ of 15 minutes. As shown in Section 3.3, the natural time scale of availability fluctuation in the wired network is hours, not minutes. Thus, with a 15 minute sampling granularity, delta fingerprints are very stable across two consecutive snapshots, with 95% of all fingerprint pairs exhibiting similarities of 0.96 or greater. Decreasing ρ results in even greater stability, which is possible since StrobeLight has a 30 second probing granularity.

The delta similarity of two *different* subnets at time t is simply the similarity of their fingerprints at t . Figure 8 depicts the pdf for cross-subnet similarity as a function of subnet size. As the subnet size grows, probability mass shifts towards the center of the similarity spectrum. However, even for subnets as small as 32 hosts, less than 2% of all subnet pairs have similarities greater than 0.8. The reason is that the various availability patterns described in Section 3.3 are randomly scattered throughout each subnet. For example, even though most subnets have a large set of always-on hosts, these hosts are randomly positioned throughout each subnet’s fingerprint vector. Thus, two vectors are unlikely to have high correlations in all bit positions, and each fingerprint is likely to be globally unique.

The tiny peaks along the right side of Figure 8 indicate a small probability that at any given moment, two subnets have completely equivalent fingerprints. To understand

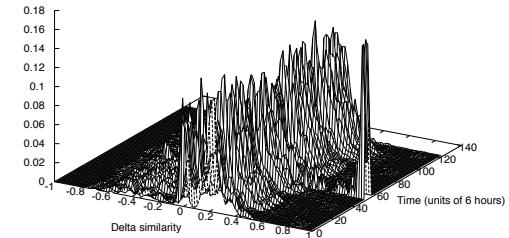
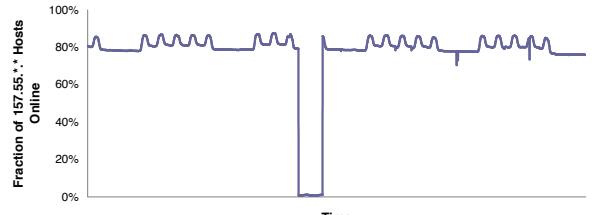
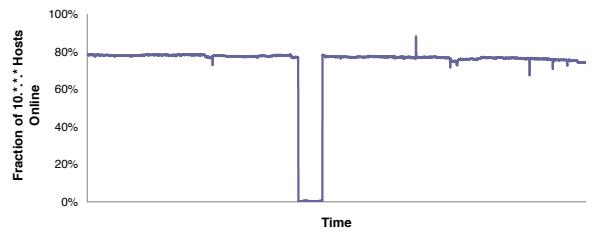


Figure 9: Temporal evolution of cross-subnet delta similarity (15 minute probe interval)



(a) Host availability in 157.55.*.*



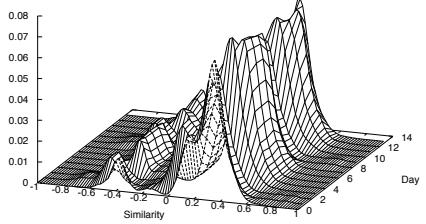
(b) Host availability in 10.*.*.*

Network anomalies during November 3 and 4 of 2005 caused the spike in fingerprint similarity seen in Figure 9.

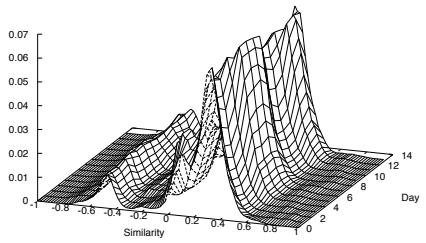
Figure 10: Punctuated availability disruptions

the origin of these peaks, we plotted cross-subnet similarity as a function of time. Figure 9 indicates a large spike in fingerprint similarity during the middle of the trace period. As Figure 10 shows, this spike was synchronous with a dramatic availability drop in several IP blocks during November 3 and 4 of 2005. When these blocks went offline, their fingerprint vectors transitioned to an “all-zeros” state, leading to an immediate increase in cross-subnet similarity.

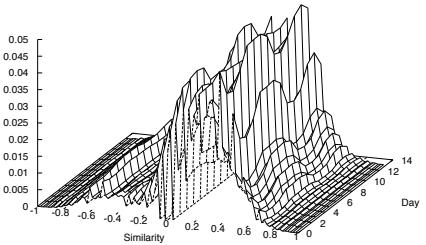
Once the anomaly terminated, the similarity distribution returned to a steady state in which all fingerprints were distinguishable. Thus, during the whole trace period, the global uniqueness property was only violated during the severe network disturbance. We return to the issue of anomaly detection in Section 4.



(a) 256 hosts per subnet, 24 hour window, 32-bit float per host



(b) 256 hosts per subnet, 24 hour window, 1-bit float per host



(c) 32 hosts per subnet, 24 hour window, 1-bit float per host

Figure 11: Cross-subnet similarity for wired and wireless subnets (24 hour window)

3.4.2 Fingerprinting Over Larger Windows

As currently described, a fingerprint is a bit vector representing the instantaneous availability of a set of hosts. In this section, we briefly describe how to extend our fingerprints to cover longer observation periods.

Cross-subnet Similarity: To create a fingerprint which covers a longer time window, we can associate each host with a floating point number instead of a single bit. Each float represents the mean availability of a host during the time period of interest. To compute the similarity between two floating point fingerprints, we examine

each pair of corresponding floats and calculate the absolute magnitude of their difference. We sum these absolute magnitudes, divide by the subnet size, and then normalize the result to the range $[-1, 1]$.

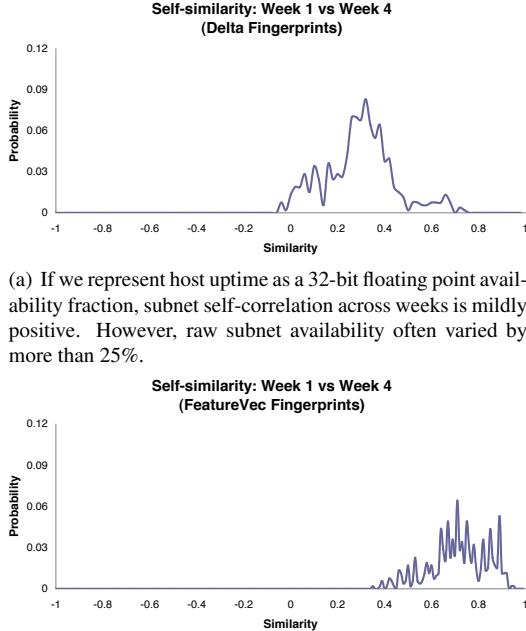
Figure 11(a) shows the temporal evolution of cross-subnet similarity using a day-long window; the subnet size was 256 hosts and each host was associated with a 32-bit floating point number. Comparing Figure 11(a) to Figure 8, we see that lengthening the fingerprint window does not change the fundamental distribution of subnet similarity. Most subnets are weakly similar or weakly dissimilar, but almost none are very similar or very dissimilar.

Figure 11(b) depicts cross-subnet similarity using a 24 hour window and “1-bit floats.” In this scenario, a fingerprint contained a single bit for each host; the bit was 1 if the host was majority-online during the window and 0 if it was majority-offline. Comparing Figure 11(a) to 11(b), we see that using these truncated floats has little impact on the similarity distribution. Even if we decrease the subnet size to 32 hosts, Figure 11(c) shows that 1-bit floats provide enough resolution to keep the likelihood of perfect cross-subnet similarity well below 1%.

Using 1-bit floats, very little storage space is needed to maintain longitudinal fingerprint databases. For example, suppose that one needs to store fingerprints for a network containing 250,000 hosts. Using 1-bit floats, an individual snapshot would consume 250,000 bits (roughly 30 KB). Assuming a 24 hour window, a full year of data will only require 11 MB of storage space.

Self-similarity: Most subnets exhibit diurnal uptime. However, the true period of their availability is a week, not a day, since availability during the weekend lacks diurnal fluctuation and is depressed relative to that of the work week. Thus, if we examine subnet self-similarity using a day-long window, there are discontinuities during the transitions into and out of the weekend. However, one might expect self-similarity to be high using a week-long window, since this window size would precisely capture a full cycle of the seven day availability pattern.

Figure 12(a) shows the distribution of wired subnet self-similarity between the first and fourth weeks of our observation period. Although self-similarity was almost always positive, the correlation was unexpectedly weak, with the bulk of the probability mass residing between 0.0 and 0.5. This surprised us, since we had predicted that a host’s availability fraction would not change much across weeks. Confronted with these results, we generated a new hypothesis, predicting that a host’s availability *class* would vary less than its availability *fraction*. For example, the uptime fraction of a long-stretch host might vary between weeks, but its availability would be unlikely to transition from long-stretch behavior to (say) diurnal behavior.



(a) If we represent host uptime as a 32-bit floating point availability fraction, subnet self-correlation across weeks is mildly positive. However, raw subnet availability often varied by more than 25%.

(b) Self-correlation is higher if we represent host availability using a 2-bit enumeration type {ALWAYS-ON, ALWAYS-OFF, DIURNAL, OTHER} and check for behavioral equivalence amongst corresponding fingerprint entries.

Figure 12: Wired subnet self-similarity using week-long windows

To test this hypothesis, we devised a new type of fingerprint called a feature vector fingerprint. Instead of associating each host with a floating point availability fraction, we gave each host a 2-bit identifier representing whether it was always-on, always-off, diurnal, or “other” (either long-stretch or unstable). We defined the similarity between two feature vectors as the number of corresponding positions with equivalent feature identifiers. As before, we divided this number by the vector size and normalized it to the range $[-1, 1]$.

Figure 12(b) confirmed our hypothesis that, at the granularity of individual hosts, availability classes are more stable than availability fractions. However, subnet self-similarity was still lower than expected given the observed stability of weekly availability cycles at the subnet level. This topic remains an important area for future research.

4 Application 2: Detecting IP Hijacking

The Internet is composed of individual administrative domains called autonomous systems (ASes). The Border Gateway Protocol (BGP) stitches these independent domains together to form a global routing system [16]. Packets follow intra-domain routing rules until they hit an inter-AS border, at which point BGP data determines the next AS that will be traversed.

As currently described, StrobeLight detects intra-AS anomalies. For example, in Section 3.4.1, we showed how StrobeLight discovered the unreachability of several large subnets from within our corporate network. In this section, we describe how to detect BGP anomalies which affect subnet visibility from the perspective of external ASes. To detect such anomalies, we must deploy StrobeLight servers *outside* of the local domain. We describe the architecture for such a system and evaluate it using Planetlab experiments and simulations driven by our corporate availability trace.

4.1 Overview of IP Hijacking

An AS declares ownership of an IP prefix through a BGP announcement. This announcement is recursively propagated to neighboring ASes, allowing each domain to determine the AS chain which must be traversed to reach a particular Internet address. BGP updates are also generated when parts of a route fail or are restored. Since BGP does not authenticate routing updates, an adversary can fraudulently declare ownership of someone else’s IP prefix and convince routers to deliver that prefix’s packets to attacker-controlled machines. An attacker can also hijack a prefix by claiming to have an attractively short route to that prefix.

Zheng *et al* describe three basic types of hijacking attack [38]. In a *blackhole attack*, the hijacker simply drops every packet that he illegitimately receives. In an *imposture attack*, the hijacker responds to incoming traffic, trying to imitate the behavior of the real subnet. In an *interception attack*, the hijacker forwards packets to their real destination, but he may inspect or manipulate the packets before forwarding them.

Due to vagaries in the BGP update process, the attacker’s fraudulent advertisement may not be visible to the entire Internet. This means that during the hijack, some ASes may route traffic to the legitimate prefix although others will not [6]. If the hijack causes divergence in external views of the prefix’s availability, we can detect the attack by deploying multiple StrobeLight servers at topologically diverse locations.

For all but the least available subnets, a blackhole attack will create a dramatic instantaneous change in externally measured fingerprints. Fingerprint deviations may be less dramatic during an imposture attack; however, as we show in Section 4.3, two arbitrary subnets are still dissimilar enough to make imposture detection easy. Interception attacks cannot be detected through fingerprint deviations since the attacker will forward StrobeLight’s probes to the real hosts in the target prefix. However, we describe a preliminary scheme in Section 4.4 that uses carefully chosen probe TTLs to detect such interceptions.

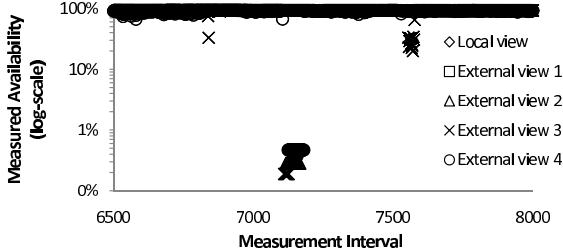


Figure 13: Availability of live IPs from different views

In our distributed StrobeLight system, the individual StrobeLight servers do not need to reside within the core routing infrastructure—they merely need to be deployed outside of the AS that they monitor. Furthermore, since anomalies are defined with respect to local measurements, there is little need for communication between the individual servers. Thus, a distributed StrobeLight system should be easy to deploy and maintain.

4.2 Does the Wide-Area Distort Probing?

As shown in Figure 7, a subnet’s fingerprint changes very slowly under normal network conditions. However, that conclusion was derived from the perspectives of vantage points *inside* the enterprise. To detect BGP anomalies, StrobeLight servers must be deployed at *external* locations. This exposes probes to the vagaries of wide-area traversal, possibly increasing delay or loss in a way that destroys fingerprint stability during non-anomalous regimes.

To determine whether fingerprints could provide diagnostic power across the wide area, we deployed StrobeLight servers on 10 topologically diverse Planetlab hosts. From April to July of 2008, these servers probed 45 Class C subnets belonging to the computer science department at the University of Michigan. We also deployed a StrobeLight server inside the local campus domain. Each server launched a probe sweep every 30 seconds, similar to our deployment inside the Microsoft corporate network. The campus network contained roughly 11,000 live IP addresses. Figure 13 shows the measured availability of these addresses from the local perspective and those of four representative Planetlab hosts. Availability was always greater than 90% from the local vantage point. This was also true for the first two external views. The third and fourth views were measured from servers that were heavily loaded with other Planetlab experiments. Processor and network utilization were consistently high on these hosts; particularly severe spikes caused the StrobeLight servers to miss incoming probe responses and underestimate true domain availability by up to 80%. However, these incidents were rare, and would not arise in a real StrobeLight deployment that used dedicated probing machines.

Near time step 7100, external views 2, 3, and 4 were almost completely partitioned from the campus domain. This partition was caused by a switchgear failure at a Detroit Edison power plant that caused punctuated router failures throughout southeast Michigan. Interestingly, this event simulated a selective blackhole attack—although views 2, 3, and 4 were cut off from the local domain, view 1 enjoyed continuous connectivity. Thus, the Planetlab deployment showed two things. First, wide-area network effects do not destroy the diagnostic utility of availability fingerprints. Second, StrobeLight can detect blackhole attacks if probe servers are deployed at topologically diverse locations.

4.3 Imposture Attacks

Blackhole attacks are not subtle. In such an attack, the adversary drops all traffic destined for the target network, creating dramatic decreases in subnet availability and thus dramatic changes in subnet fingerprints. Imposture attacks are potentially more difficult to detect, since the adversary seeks to *mimic* the behavior of hosts in the target domain. In particular, we are interested in detecting spectrum agility attacks, first described by Ramachandran and Feamster [31]. The goal of a spectrum attack is to elude IP-based blacklists using short-lived manipulations of BGP state. Spammers hijack a large network, e.g., a /8 prefix, send a few pieces of spam from random IP addresses within the prefix, and then withdraw the fraudulent BGP advertisement a few minutes later. By using short-lived routing advertisements, spammers increase the likelihood that their hosts will be unreachable by the time that white hat forensics begin. By sending a small amount of traffic from each host, and by randomly scattering the traffic throughout a large address space, spammers avoid filtering by DNS-based blacklists [21].

To determine whether StrobeLight can detect spectrum attacks, we used simulations driven by availability data from the Microsoft network. We used this trace data instead of the Michigan data because it contained many more IP addresses, and spectrum attacks require large address spaces for maximum effectiveness. Our simulations used a trace gathered between July 29, 2006 and September 1, 2006. To include the largest possible host set in our evaluation, we did not filter hosts based on their unknown fraction. During this observation period, we saw 238,951 unique IP addresses. Our simulations examined the largest subnets demarcated by standard Class A/B/C rules. We also examined a “mega” subnet consisting of all IP addresses in the trace.

During each simulation run, we iterated through our availability data in strides of 15 minutes; during each iteration, we compared each subnet’s fingerprint to that of a similarly sized attacker subnet in which a random fraction of hosts responded to StrobeLight’s pings. StrobeLight

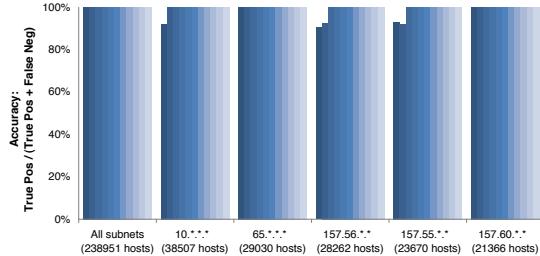


Figure 14: Detecting spectrum agility attacks

detected the spectrum attack if the similarity of the two fingerprints was beneath a threshold c . More specifically, let $f_{real,t}$ represent the fingerprint of a real subnet at time t and $f_{fake,t}$ be the fingerprint of the attacker subnet. Let $sim()$ compute the similarity of two fingerprints. Given a similarity cutoff c , we define StrobeLight’s detection accuracy at time t as follows:

- **True positive:** $sim(f_{real,t-1}, f_{fake,t}) < c$. The attacker’s fake subnet at time t is too dissimilar to the real subnet’s fingerprint from the previous timestep. StrobeLight raises an alarm in this case.
- **False negative:** $sim(f_{real,t-1}, f_{fake,t}) \geq c$. The fake subnet is sufficiently similar to the real subnet that StrobeLight does not raise an alarm.

Every simulated comparison should raise an alarm, so there are no true negatives or false positives.

Figure 14 shows StrobeLight’s detection accuracy in the five largest subnets. We also show results for an attack against the “mega-subnet” containing all hosts, since this is the best that we can approximate a large /8 prefix. Each cluster of bars represents detection accuracy for a specific subnet. Within a cluster, the i -th bar is our detection accuracy when a random $i * 10\%$ of hosts in the attacker subnet respond to probes. For all of the results, we used a similarity cutoff c of 0.78; this value minimized the false negative rate.

In the mega-subnet containing 238,951 hosts, StrobeLight had perfect detection accuracy across all time steps. StrobeLight also had perfect accuracy for two of the five classful subnets. In the other three, detection accuracy for low response fractions dipped as low as 90%. These subnets were affected by a DNS failure which caused their hosts to spend part of the observation period in an unknown state. StrobeLight assumes that unknown hosts are offline, so an attacker could hijack these subnets during the DNS failure and evade detection by rarely responding to StrobeLight pings. However, StrobeLight would raise alarms at the beginning of the DNS anomaly, since a large number of hosts would appear to go offline suddenly. Thus, human operators would be more vigilant for additional problems during this time period. In Section 4.6, we return to the issue of StrobeLight’s reliance on DNS infrastructure.

If an attacker could measure availability trends in our subnets, he could mimic the legitimate distribution of probe responses during the spectrum attack and avoid detection by StrobeLight. However, many organizations already perform ingress filtering of ping probes destined for internal hosts, eliminating the most obvious way for an adversary to collect availability data.

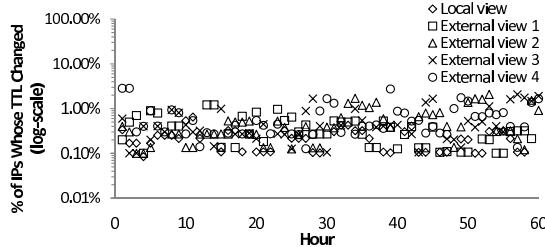
The attacker could try to spoof the IP address of a real StrobeLight server, and use the spoofed address to launch surveillance probes. There are several ways to deal with such an attack. One simple solution is to have the legitimate StrobeLight servers periodically audit each other using a shared-secret challenge/response protocol. If an attacker spoofs server S_0 ’s address, and the spoof is visible by another server S_1 , the fake S_0 will fail S_1 ’s challenge, and S_1 can raise an alarm.

4.4 Interception Attacks

In an interception attack, the adversary convinces routers to send other people’s traffic through attacker-controlled machines. These machines may inspect or tamper with the packets before forwarding them to their real destination. The current version of StrobeLight cannot detect such interceptions, since the interceptor does not drop legitimate probe packets or generate false probe responses. We have preliminary thoughts about how to modify StrobeLight to detect interceptions, and we briefly sketch some ideas below. However, a full exploration is left to future work.

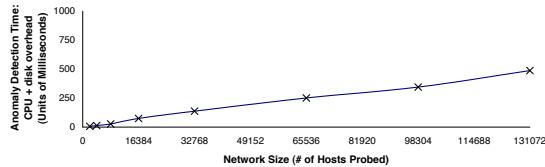
Since two arbitrary prefixes are likely to be topologically distant [38], an interception attack that affects a StrobeLight probing path should *lengthen* the route between the StrobeLight server and the monitored prefix. In theory, this will increase the latency from the server to the monitored prefix. So, the server can raise an alarm if it detects a correlated spike in response latencies across all prefix hosts. Unfortunately, latency may display significant jitter during non-anomalous conditions, so a naive implementation of this scheme will generate excessive false alarms.

Instead of looking for latency changes, StrobeLight could look for hop count changes. Previous research has shown that the hop count between two arbitrary prefixes is stable in the short to medium term [36, 38]. We verified this result with our StrobeLight deployment at the University of Michigan. Figure 15 shows the stability of hop counts from the internal Michigan server and from several external vantage points. Both internal and external servers recalculated their hop count to Michigan hosts once an hour; these recalculations were staggered across each hour. Recalculations typically resulted in TTL changes for less than 1% of all nodes, and we believe that most changes were due to lost tracing packets instead of actual host movement within the target domain.



Stub networks rarely change their location with respect to the network core. Thus, the hop counts between hosts in that stub and an external vantage point are stable.

Figure 15: Hop count stability



Each data point represents the average of 100 trials. Standard deviations were very small.

Figure 16: Scalability of Analysis Engine

Since interception attacks are likely to lengthen the route between a StrobeLight server and its target prefix, they are detectable by monitoring the hop count between the target prefix and the distributed measurement sites. This idea was first proposed by Zheng *et al* [38], and a variant could be integrated into StrobeLight. Each server would carefully set the TTLs of its probes to the expected hop count to the target prefix. A sudden increase in this path length will cause the probes to be dropped before they reach their destination; the StrobeLight server will perceive this as a sudden decrease in prefix availability and raise an alarm. This solution is more attractive than the latency-based scheme since hop counts are much more stable than latency. However, the hop count technique assumes that the attacker has limited topological knowledge. In particular, if the interceptor knows the routes connecting the target prefix, the StrobeLight servers, and the interceptor’s routers, he can rewrite TTLs in a straightforward way to elude detection.

4.5 Performance

Anomaly detection consists of three steps: issuing the ping sweep from the probe machine, transferring the probe results to the analysis machine, and performing fingerprint calculations on the analysis machine. The first step is the slowest one, since we spread the probing sweep over several seconds to avoid noticeable network spikes. The second step should be fast even if the probing ma-

chine is different than the analysis machine, since probe results are just small bit vectors. As shown in Figure 16, the final calculation step is also fast. Figure 16 shows that once the analyzer has pulled the ping results onto local storage, the time needed to calculate new fingerprints and perform threshold calculations is less than half a second, even for networks with 130,000 hosts.

4.6 Discussion

StrobeLight queries DNS servers to determine which IP addresses to probe. Depending on one’s perspective, this is a vice or a virtue. StrobeLight’s sensitivity to DNS state means that it can detect some anomalies in DNS operation. However, this opens StrobeLight to DNS-mediated attacks in which adversaries try to disrupt StrobeLight’s DNS fetches before tampering with BGP state. The IP prefixes owned by an enterprise are fairly stable, so we could manually configure StrobeLight with these prefixes and probe every address without regard to whether it was assigned internally (in fact, this is what we did for the StrobeLight deployment at the University of Michigan, since we lacked access to the DNS zone files). The penalty would be an increase in the prober’s network load; also, if there are many unassigned addresses, cross-subnet similarity will naturally be higher, leading to more false alarms.

5 Related Work

Several commercial products provide enterprise-scale network monitoring without requiring end-host modification. For example, in the SiteScope system [17], a centralized server remotely logs into client systems and reads local performance counters. Tools like this collect a wider variety of data than StrobeLight, which only measures availability. However, StrobeLight can scan more machines per second, since it uses simple ping probes instead of comparatively heavyweight remote logins. StrobeLight is also easier to deploy in heterogeneous end-host environments, since ICMP probes work “out-of-the-box” across all commodity operating systems, but remote login procedures can differ substantially across OSes.

Passive introspection of preexisting traffic can be used to infer path characteristics or host availability. For example, Padmanabhan *et al* record the end-to-end loss rate inside a client-server flow and use Bayesian statistics to extrapolate loss rates for interior IP links [28]. Passive detection of host availability is attractive for two reasons. First, it does not generate new traffic. Second, explicit probing may trigger intrusion detection systems on leaf networks, a problem occasionally encountered with active probing systems deployed on PlanetLab [34]. Despite these advantages, passive probing was ill-suited for our

goal of tracking per-host availability in a large network. The time that a host is online is a superset of the time that it is generating network traffic, so passive observations of per-host packet flows may underestimate true availability. Also, a key design goal was to minimize the new infrastructure that had to be pushed to end hosts or the corporate routing infrastructure. Installing custom network introspection code on every end host was infeasible. Placing such code inside the core network infrastructure was also infeasible due to the complex web of proxies, firewalls, and routers that would have to be instrumented to get a full view of each host’s network activity.

Most prior work on IP hijack detection has required modification to core Internet routers. Some systems require routers to perform cryptographic operations to validate BGP updates [2, 12, 19], whereas others require changes to router software to make BGP updates more robust to tampering [35, 37]. We eschewed such designs due to the associated deployment problems.

Several systems use passive monitoring of BGP dynamics to detect inconsistencies in global state [22, 23, 33]. These systems typically search for anomalies in one or more publicly accessible databases such as RouteViews [27], which archives BGP state from multiple vantage points, or the Internet Routing Registry [3], which contains routing policies and peering information for each autonomous system. Passive monitoring eases deployability concerns. However, data freshness becomes a concern when dealing with “eventually updated” repositories such as the IRR, and even RouteViews data is only updated once every two hours. Legitimate changes to routing policy may also be indistinguishable from hijacking attacks in terms of BGP semantics, making disambiguation difficult in some cases. In contrast, if our availability fingerprints indicate that a large chunk of hosts have suddenly gone offline or changed their availability profile, it is extremely unlikely that this is a natural phenomenon.

Hu and Mao were the first to use data plane fingerprints in the context of hijack detection [18]. In their system, a live BGP feed is monitored for suspicious updates. If an IP prefix is involved in a questionable update, its hosts are scanned from multiple vantage points using nmap OS fingerprinting [15], IP ID probing [7], and ICMP timestamp probing [18]. The results are presented to a human operator who determines if they are inconsistent. Our system differs in three ways. First, we do not require privileged access to a live BGP feed, easing deployability. Second, we continually calculate subnet fingerprints, whereas Hu’s system only calculates fingerprints upon detecting suspicious BGP behavior, behavior which may take several minutes to propagate to a particular vantage point. Third, we can finish a probing sweep in less than 30 seconds, whereas several of Hu’s scans may take several minutes to complete. Given the short-lived nature of spectrum

agility attacks [31], we believe that quick, frequent scanning is preferable, if only to serve as a tripwire to trigger slower, “deeper” scans.

Zheng *et al* detect hijacking attacks by measuring the hop count from monitor hosts to the IP prefixes of interest [38]. For each prefix, the monitor selects a reference point that is topologically close to the prefix and lies along the path from the monitor to the prefix. In normal situations, the hop count along the monitor-reference point path should be close to that of the monitor-prefix path. When the prefix is hijacked, the hop count along the two paths should diverge. Zheng’s system avoids the deployability problems mentioned above, since hop counts can be determined by any host that can run traceroute. However, the system assumes that a reference point can be found which is immediately connected to the target prefix and responds to ICMP messages; if the reference point is further out, the hijacker can hide within the extra hops. Our system only requires that end hosts respond to pings. Furthermore, our system tracks the availability of individual hosts, whereas Zheng’s system only tracks the availability of a few representative hosts in each target prefix.

6 Conclusion

Many distributed systems would benefit from an infrastructure that collected high resolution availability measurements for individual hosts. Unfortunately, existing frameworks either do not scale, do not track every host in the network, or store data in such a way that makes global analysis difficult. In this paper we describe StrobeLight, an enterprise-level tool for collecting fine-grained availability data. Our current prototype has measured the up-time of hundreds of thousands of hosts in our corporate network for almost two years. Using the longitudinal data generated by this tool, we performed extensive analyses of availability in our wired and wireless networks. Using external Planetlab deployments and simulations, we also demonstrated how StrobeLight’s real-time analysis engine can detect wide-area network anomalies. Our operational experiences indicate that StrobeLight’s anomaly detection is fast and accurate.

References

- [1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of OSDI*, pages 1–14, Boston, MA, December 2002.
- [2] W. Aiello, J. Ioannidis, and P. McDaniel. Origin Authentication in Interdomain Routing. In *Proceedings of CCS*, pages 165–178, Washington, DC, October 2003.
- [3] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing Policy Specification Language (RPSL). RFC 2622, June 1999.

- [4] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of SOSP*, pages 131–145, Banff, Canada, October 2001.
- [5] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Pittsburgh, November 2004.
- [6] H. Ballani, P. Francis, and X. Zhang. A Study of Prefix Hijacking and Interception in the Internet. In *Proceedings of SIGCOMM*, pages 265–276, Kyoto, Japan, August 2007.
- [7] S. Bellovin. A Technique for Counting NATted Hosts. In *Proceedings of the SIGCOMM Internet Measurement Workshop*, pages 267–272, Marseille, France, November 2002.
- [8] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd IPTPS*, Berkeley, CA, February 2003.
- [9] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: system support for automated availability management. In *Proceedings of NSDI*, pages 337–350, March 2004.
- [10] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of ACM SIGMETRICS*, pages 34–43, Santa Clara, CA, June 2000.
- [11] W. Bolosky, J. Douceur, and J. Howell. The Farsite Project: A Retrospective. *ACM SIGOPS Operating Systems Review*, 41(2):17–26, April 2007.
- [12] K. Butler, P. McDaniel, and W. Aiello. Optimizing BGP Security by Exploiting Path Stability. In *Proceedings of CCS*, pages 298–310, Alexandria, VA, November 2006.
- [13] D.-F. Chang, R. Govindan, and J. Heidemann. Locating BGP Missing Routes Using Multiple Perspectives. In *Proceedings of the SIGCOMM Workshop on Network Troubleshooting*, pages 301–306, Portland, OR, September 2004.
- [14] P. Cincotta, M. Mendez, and J. Nunez. Astronomical Time Series Analysis I: A Search for Periodicity Using Information Entropy. *The Astrophysical Journal*, 449:231–235, August 1995.
- [15] Fyodor. nmap security scanner. <http://insecure.org/nmap/>.
- [16] IETF IDR Working Group. A Border Gateway Protocol 4 (BGP-4). RFC 1771, March 1995.
- [17] Hewlett-Packard Development Company. HP SiteScope software: Data sheet. White paper, August 2008.
- [18] X. Hu and Z. Morley Mao. Accurate Real-time Identification of IP Prefix Hijacking. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 3–17, Oakland, California, May 2007.
- [19] Y.-C. Hu, A. Perrig, and M. Sirbu. SPV: Secure Path Vector Routing for Securing BGP. In *Proceedings of SIGCOMM*, pages 179–192, Portland, OR, September 2004.
- [20] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of VLDB*, pages 321–332, Berlin, Germany, September 2003.
- [21] J. Jung and E. Sit. An Empirical Study of Spam Traffic and the Use of DNS Black Lists. In *Proceedings of IMC*, pages 370–375, Taormina, Sicily, Italy, October 2004.
- [22] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur. Topology-based Detection of anomalous BGP messages. In *Proceedings of RAID*, pages 17–35, Pittsburgh, PA, September 2003.
- [23] M. Lad, D. Massey, D. Pei, Y. Wu, B. Zhang, and L. Zhang. Phas: A Prefix Hijack Alert System. In *Proceedings of USENIX Security*, pages 153–166, Vancouver, Canada, August 2006.
- [24] H. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iplane: An Information Plane for Distributed Services. In *Proceedings of OSDI*, pages 367–380, Seattle, WA, November 2006.
- [25] J. Mickens and B. Noble. Exploiting Availability Prediction in Distributed Systems. In *Proceedings of NSDI*, pages 73–86, San Jose, CA, May 2006.
- [26] J. Mickens and B. Noble. Concilium: Collaborative Diagnosis of Broken Overlay Routes. In *Proceedings of DSN*, pages 225–234, Edinburgh, UK, June 2007.
- [27] University of Oregon. *Route Views Project*. <http://www.routeviews.org>.
- [28] V. Padmanabhan, L. Qiu, and H. Wang. Passive Network Tomography Using Bayesian Inference. In *Proceedings of SIGCOMM Internet Measurement Workshop*, pages 93–94, Marseille, France, November 2002.
- [29] K. Park and V. Pai. CoMon: A mostly-scalable monitoring system for PlanetLab. *Operating Systems Review*, 40(1):65–74, January 2006.
- [30] S. Pincus. Approximate entropy as a measure of system complexity. In *Proceedings of the National Academy of Science*, pages 2297–2301, USA, March 1991.
- [31] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *Proceedings of SIGCOMM*, pages 291–302, Pisa, Italy, September 2006.
- [32] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.
- [33] G. Siganos and M. Faloutsos. Neighborhood Watch for Internet Routing: Can We Improve the Robustness of Internet Routing Today? In *Proceedings of INFOCOM*, pages 1271–1279, Anchorage, AK, May 2007.
- [34] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using Planetlab for Network Research: Myths, Realities, and Best Practices. In *Proceedings of WORLDSPAN*, pages 67–72, San Francisco, CA, December 2005.
- [35] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. H. Katz. Listen and Whisper: Security Mechanisms for BGP. In *Proceedings of NSDI*, pages 127–140, San Francisco, CA, March 2004.
- [36] R. Teixeira, S. Agarwal, and J. Rexford. BGP Routing Changes: Merging Views from Two ISPs. In *SIGCOMM Computer Communications Review*, pages 79–82, October 2005.
- [37] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang. Detection of Invalid Routing Announcement in the Internet. In *Proceedings of DSN*, pages 59–68, Bethesda, MD, June 2002.
- [38] C. Zheng, L. Ji, D. Pei, J. Wang, and P. Francis. A Light-Weight Distributed Scheme for Detecting IP Prefix Hijacks in Real-Time. In *Proceedings of SIGCOMM*, pages 277–288, Kyoto, Japan, August 2007.

Hashing Round-down Prefixes for Rapid Packet Classification

Fong Pong

Broadcom Corp.

2451 Mission College Blvd., Santa Clara, CA 95054
fpong@broadcom.com

Nian-Feng Tzeng

Center for Advanced Computer Studies

University of Louisiana at Lafayette, LA 70504
tzeng@cacs.louisiana.edu

Abstract — Packet classification is complex due to multiple fields present in each filter rule, easily manifesting itself as a router performance bottleneck. Most known classification approaches involve either hardware support or optimization steps (to add precomputed markers and insert rules in the search data structures). Unfortunately, an approach with hardware support is expensive and has limited scalability, whereas one with optimization fails to handle incremental rule updates effectively. This work treats a rapid packet classification mechanism, realized by hashing round-down prefixes (HaRP) in a way that the source and the destination IP prefixes specified in a rule are rounded down to “designated prefix lengths” (DPL) for indexing into hash sets. Utilizing the first ζ bits of an IP prefix with l bits (for $\zeta \leq l$, $\zeta \in \text{DPL}$) as the key to the hash function (instead of using the original IP prefix), HaRP exhibits superb hash storage utilization, able to not only outperform those earlier software-oriented classification techniques but also well accommodate dynamic creation and deletion of rules. HaRP makes it possible to hold all its search data structures in the local cache of each core within a contemporary processor, dramatically elevating its classification performance. Empirical results measured on our Broadcom BCM-1480 multicore platform under nine filter datasets obtained from a public source unveil that HaRP enjoys up to some 5 \times (or 10 \times) throughput improvement when compared with well-known HyperCuts (or Tuple Space Search).

1 Introduction

Packet classification is basic to a wide array of Internet applications and services, performed at routers by applying “rules” to incoming packets for categorizing them into flows. It employs multiple fields in the header of an arrival packet as the search key for identifying the best suitable rule to apply. Rules are created to differentiate packets based on the values of their corresponding header fields, constituting a filter set. Header fields may contain network addresses, port numbers, the protocol type, TCP flags, ICMP message type and code number, VLAN tags, DSCP and 802.1p codes, etc. A field value in a filter can be an IP prefix (e.g., source or destination sub-network), a range (e.g., source or destination port numbers), or an exact number (e.g., protocol type or TCP flag). A real filter dataset often contains multiple rules for a pair of communicating networks, one for each application. Similarly,

an application is likely to appear in multiple filters, one for each pair of communicating networks using the application. Therefore, lookups over a filter set with respect to multiple header fields are complex [9] and often become router performance bottlenecks.

Various classification mechanisms have been considered, and they aim to quicken packet classification through hardware support or the use of specific data structures to hold filter datasets (often in SRAM and likely with optimization) for fast search [25]. Hardware support frequently employs FPGAs (field programmable gate arrays) or ASIC logics [4, 21], plus TCAM (ternary content addressable memory) to hold filters or registers for rule caching [8]. Key design goals with hardware support lie in simple data structures and search algorithms to facilitate ASIC or FPGA implementation and low storage requirements to reduce the TCAM costs. They tend to prevent a mechanism with hardware support from handling incremental rule updates efficiently, and any change to the mechanism (in its search algorithm or data structures) is usually expensive. Additionally, such a mechanism exhibits limited scalability, as TCAM employed to hold a filter set dictates the maximal set size allowable. Likewise, search algorithms dependent on optimization via preprocessing (used by recursive flow classification [9]) or added markers and inserted rules (stated in rectangle tuple space search (TSS) [24], binary TSS on columns [28], diagonal-based TSS [15], etc.) for speedy lookups often cannot deal with incremental rule updates effectively. A tuple under TSS specifies the involved bits of those fields employed for classification, and probes to tuple space for appropriate rules are conducted via fast exact-match search methods like hashing.

Many TSS-based classifiers employ extra SRAM (in addition to processor caches). Unlike TCAM, SRAM costs far less and consumes much lower energy. Further, if the required SRAM size is made small to fit in an on-chip module, the cost incurred for the on-chip SRAM can be very low, since it shares the same fabrication processes as those for on-chip caches. However, the inherent limitation of a TSS classifier in dealing with incremental rule updates (deemed increasingly common due to such popular applications as voice-over-IP, gaming, and video conferencing, which all involve dynamically triggered insertion and removal of rules in order for the firewall to handle packets properly) will soon become a major concern [30].

This article treats hashing round-down prefixes (HaRP) for rapid packet classification, where an IP prefix with l bits is rounded down to include its first ζ bits only (for $\zeta \leq l$, ζ

\in DPL, “designated prefix lengths” [17]). With two-staged search, HaRP achieves high classification throughput and superior memory efficiency by means of (1) rounding down prefixes to a small number of DPL (denoted by m , i.e., m possible designated prefix lengths), each corresponding to one hash unit, for fewer (than 32 under IPv4, when every prefix length is permitted without rounding down) hash accesses per packet classification, and (2) collapsing those hash units to one lumped hash (LuHa) table for better utilization of table entries, which are set-associative. Based on a LuHa table keyed by the source and destination IP prefixes rounded down to designated lengths, HaRP not only enjoys fast classification (due to a small number of hash accesses) but also handles incremental rule updates efficiently (without precomputing markers or inserting rules often required by typical TSS). While basic HaRP identifies up to two candidate sets in the LuHa table to hold a given filter rule, generalized HaRP (denoted by HaRP*) may store the rule in any one of up to $2m$ candidate sets, considerably elevating table utilization to lower the probability of set overflow and achieving good scalability even for a small set-associative degree (say, 4). Each packet classification under HaRP* requires to examine all the possible $2m$ candidate sets (*in parallel* for those without conflicts, i.e., those in different memory modules which constitute the LuHa Table), where those sets are identified by the hash function keyed with the packet’s source and destination IP addresses, plus their respective round-down prefixes. HaRP is thus to exhibit fast classification, due to its potential of parallel search over candidate sets. With SRAM for the LuHa table and the application-specific information table (for holding filter fields other than source and destination IP prefixes), HaRP exhibits a lower cost and better scalability than its hardware counterpart. With its required SRAM size dropped considerably (to some 200KB at most for all nine filter datasets examined), HaRP makes it possible to hold all its search data structures in the local cache of a core within a contemporary processor, further boosting its classification performance.

Our LuHa table yields high storage utilization via identifying multiple candidate sets for each rule (instead of just a single one under a typical hash mechanism), like the earlier scheme of d -left hashing [1]. However, the LuHa table differs from d -left hashing in three major aspects: (1) the LuHa table requires just one hash function, as opposed to d functions needed by d -left hashing (which divides storage into d fragments), one for each fragment, (2) the hash function of the LuHa table under HaRP* is keyed by $2m$ different prefixes produced from each pair of the source and the destination IP addresses, and (3) a single LuHa table obtained by collapsing separate hash units is employed to attain superior storage utilization, instead of one hash unit per prefix length to which d -left hashing is applied.

Extensive evaluation on HaRP has been conducted on our platform comprising a Broadcom’s BCM-1480 SoC (System on Chip) [18], which has four 700MHz SB-1™ MIPS cores [12], under nine filter datasets obtained from a public source [29]. The proposed HaRP was made multithreaded so that up to 4 threads could be launched to take advantage of the 4 SB-1™ cores for gathering real elapsed times via the BCM-1480 ZBus counter, which ticks at every system clock. Measured throughput results of HaRP are compared with those of its various counterparts (whose source codes were downloaded from a public source [29] and then made multithreaded for) executing on the same platform to classify millions of packets generated from the traces packaged with the filter datasets. Our measured results reveal that HaRP* boosts classification throughput by some 5× (or 10×) over well-known HyperCuts [20] (or Tuple Space Search [24]), when its LuHa table has a total number of entries equal to $1.5n$ and there are 4 designated prefix lengths, for a filter dataset sized n . HaRP attains superior performance, on top of its efficient support for incremental rule updates lacked by previous techniques, making it a highly preferable software-based packet classification technique.

2 Pertinent Work and Tuple Space Search

Packet classification is challenging and its cost-effective solution is still in pursuit actively. Known classification lookup mechanisms may be categorized, in accordance with their implementation approaches, as being hardware-centric and software-oriented, depending upon if dedicated hardware logics or specific storage components (like TCAM or registers) are used. Different hardware-centric classification mechanisms exist. In particular, a mechanism with additional registers to cache evolving rules and dedicated logics to match incoming packets with the cached rules was pursued [8]. Meanwhile, packet classification using FPGA was considered [21] by using the BV (Bit Vector) algorithm [13] to look up the source and destination ports and employing a TCAM to hold other header fields, with search functionality realized by FPGA logic gates. Recently, packet classification hardware accelerator design based on the HiCuts and HyperCuts algorithms [3, 20] (briefly reviewed in Section 2.1), has been presented [11]. Separately, effective methods for dynamic pattern search were introduced [4], realized by reusing redundant logics for optimization and by fitting the whole filter device in a single Xilinx FPGA unit, taking advantage of built-in memory and XOR-based comparators in FPGA.

Hardware approaches based on TCAM are considered attractive due to the ability for TCAM to hold the don’t care state and to search the header fields of an incoming packet against *all TCAM entries* in a rule set simultaneously [16, 27]. While deemed as most widely employed storage components in support of fast lookups, TCAM has such noticeable shortcomings (listed in [25]) as lower density, higher power consumption, and being pricier and unsuitable for dynamic

rules, since incremental updates usually require many TCAM entries to be shifted (unless provision like those given earlier [19, 27] is made). As a result, software-oriented classification is more attractive, provided that its lookup speed can be quickened by storing rules in on-chip SRAM.

2.1 Software-Oriented Classification

Software-oriented mechanisms are less expensive and more flexible (better adaptive to rule updates), albeit to slower filter lookups when compared with their hardware-centric counterparts. Such mechanisms are abundant, commonly involving efficient algorithms for quick packet classification with an aid of caching or hashing (via incorporated SRAM). Their classification speeds rely on efficiency in search over the rule set (stored in SRAM) using the keys constituted by corresponding header fields. Several representative software classification techniques are reviewed in sequence.

Recursive flow classification (RFC) carries out multistage reduction from a lookup key (composed of packet header fields) to a final *classID*, which specifies the classification rule to apply [9]. Given a rule set, preprocessing is required to decide memory contents so that the sequence of RFC lookups according to a lookup key yields the appropriate *classID* [9]. Preprocessing results can be put in SRAM for fast accesses, important for RFC as it involves multiple stages of lookups. Any change to the rule set, however, calls for memory content recomputation, rendering it unsuitable for frequent rule updates.

Based on a precomputed decision tree, HiCuts (Hierarchical Intelligent Cuts) [10] holds classification rules merely in leaf nodes and each classification operation needs to traverse the tree to a leaf node, where multiple rules are stored and searched sequentially. During tree search, HiCuts relies on local optimization decisions at each node to choose the next field to test. Like HiCuts, HyperCuts is also a decision tree-based classification mechanism, but each of its tree nodes splits associated rules possibly based on multiple fields [20]. It builds a decision tree, aiming to involve the minimal amount of total storage and to let each leaf node hold no more than a predetermined number of rules. HyperCuts is shown to enjoy substantial memory reduction while considerably quickening the worst-case search time under core router rule sets [20], when compared with HiCuts and other earlier classification solutions.

An efficient packet classification algorithm was introduced [2] by hashing flow IDs held in digest caches (instead of the whole classification key comprising multiple header fields) for reduced memory requirements at the expense of a small amount of packet misclassification. Recently, fast and memory-efficient (2-dimensional) packet classification using Bloom filters was studied [7], by dividing a rule set into multiple subsets before building a crossproduct table [23] for each subset individually. Each classification search probes only those subsets that contain matching rules (and skips the rest) by means of Bloom filters, for sustained high throughput. The mean memory requirement is claimed to be some 32 ~ 45 bytes per rule. As will be

demonstrated later, our mechanism achieves faster lookups (involving 8~16 hash probes plus 4 more SRAM accesses, which may all take place in parallel, per packet) and consumes fewer bytes per rule (taking 15 ~ 25 bytes per rule).

A fast dynamic packet filter, dubbed Swift [30], comprises a fixed set of instructions executed by an in-kernel interpreter. Unlike packet classifiers, it optimizes filtering performance by means of powerful instructions and a simplified computational model, involving a kernel implementation.

2.2 Tuple Space Search (TSS)

Having rapid classification potentially (with an aid of optimization) without additional expensive hardware, TSS has received extensive studies. It embraces versatile software-oriented classification and involves various search algorithms. Under TSS, a tuple comprises a vector of k integer elements, with each element specifying the *length* or *number of bits* of a header field of interest used for the classification purpose. As the possible numbers of bits for interested fields present in the classification rules of a filter dataset tend to be small, all length combinations of the k fields constituting tuple space are rather contained [24]. In other words, while the tuple space T in theory comprises totally $\prod_{i=1,k} \text{prefix.length}(\text{field}_i)$ tuples, it only needs to search *existing tuples* rather than the entire space T .

A search key can be obtained for each incoming packet by concatenating those involved bits in the packet header. Consider a classic 5-dimensional classification problem, with packets classified by their source IP address (sip), source port number (spn), destination IP address (dip), destination port number (dpn), and protocol type (pt). An example tuple of (sip, dip, spn, dpn, pt) = (16, 24, 6, 4, 6) means that the source and the destination IP addresses are respectively a 16-bit prefix and a 24-bit prefix. The number of prefix bits used to define the tuple elements of sip and dip is thus clear. On the other hand, the port numbers and the protocol type are usually specified in ranges; for example, [1024, 2112] referring to the port number from 1024 to 2112. For TSS, those range files are (1) handled separately (like what was stated in [3]), (2) encoded by *nested level and range IDs* [24], or (3) transformed into collections of sub-ranges each corresponding to a prefix (namely, a range with an exact power of two), resulting in rule dataset expansion.

TSS Implementation Consideration

TSS intends to achieve high memory efficiency and fast lookups by exploiting a well sanctioned fact of rule construction resulting from optimization. Its optimization methods include:

1. Tuple Pruning and Rectangle Search, using *markers* and pre-computed *best-matched rules* to achieve the worst-case lookup time of $2W-1$ for two-dimensional classification, with W being the length of source and destination IP prefixes [24],
2. Binary Search on Columns, considered later [28] to reduce the worst-case lookup time down to $O(\log^2 W)$, while involving $O(N \times \log^2 W)$ memory for N rules, and

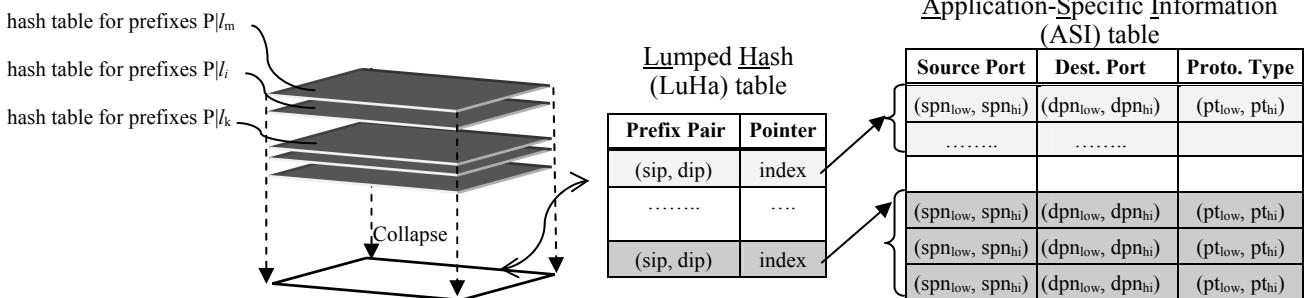


Figure 1. HaRP classification mechanism comprising one set-associative hash table (obtained by lumping multiple hash tables together) and an application-specific information table.

3. Diagonal-based Search to exhibit the search time of $O(\log W)$ for two-dimensional filters, with a large memory requirement of $O(N^2)$ [15].

While TSS (with optimization) is generally promising, it suffers from the following limitations.

Expensive Incremental Updates. Dynamic creation and removal of classification rules may prove to be challenging to those known TSS methods. However, dynamic changes to rule datasets take place more frequently going forward, due to many growing popular applications, such as voice-over-IP, gaming, and video conferencing, which all require dynamically triggered insertion and removal of rules in order for the firewall to handle packets properly. This inability in dealing with frequent rule updates is common to TSS-based packet classification, because its high search rate and efficient memory (usually SRAM) utilization result from storing contents in a way specific to contents themselves, and any change to the rule dataset requires whole memory content recomputed and markers/rules reinserted. With its nature of complex and prohibitively expensive memory management in response to rule changes, TSS is unlikely to arrive at high performance.

Limited Parallelism. TSS with search optimization lends itself to sequential search, as the next tuple to be probed depends on the search result of the current tuple. Its potential in parallelism is rather limited as the number of speculative states involved grows exponentially when the degree increases.

Extensibility to Additional Fields. Results for two-dimensional TSS have been widely reported. However, it is unclear about TSS performance when the number of fields rises (to accommodate many other fields, including TCP flags, ICMP message type and code number, VLAN tags, DSCP and 802.1p codes, besides commonly mentioned five fields), in particular, if markers and precomputation for best rules are to be applied.

3 Proposed HaRP Architecture

3.1 Fundamentals and Pertinent Data Structures

As eloquently explained earlier [25, 26], a classification rule is often specified with a pair of communicating networks, followed by the application-specific constraints (e.g., port

numbers and the protocol type). Our HaRP exploits this situation by considering the fields on communicating networks and on application-specific constraints separately, comprising two search stages. Its first stage narrows the search range via communicating network prefix fields, and its second stage checks other fields on only entries chosen in the first stage.

Basic HaRP

As depicted in Figure 1, the first stage of HaRP comprises a single set-associative hash table, referred to as the LuHa (lumped hash) table. Unlike typical hash table creation using the object key to determine one single set for an object, our LuHa table aims to achieve extremely efficient table utilization by permitting *multiple candidate sets* to accommodate a given filter rule and yet maintaining fast search over those possible sets in parallel during the classification process. It is made possible by (1) adopting *designated prefix length*, DPL: $\{l_1, l_2, \dots, l_i, \dots, l_m\}$, where l_i denotes a prefix length, such that for any prefix P of length w (expressed by $P|w$) with $l_i \leq w < l_{i+1}$, P is rounded down to $P|l_i$ before used to hash the LuHa table, and (2) storing a filter rule in the LuHa table hashed by either its source IP prefix (sip, if not wild carded) or destination IP prefix (dip, if not wild carded), after they are rounded down. Each prefix length ζ , with $\zeta \in \text{DPL}$, is referred to as a *tread*. Given P , it is hashed by treating $P|l_i$ as an input to a hash function to get a d -bit integer, where d is dictated by the number of sets in the LuHa table. Since treads in DPL are determined in advance, the numbers of bits in an IP address of a packet used for hash calculation during classification are clear and their hashed values can be obtained in parallel for concurrent search over the LuHa table. Our classification mechanism results from hashing round-down prefixes (HaRP) during both filter rule installation and packet classification search, thereby so named.

The LuHa table comprises collapsed individual hash tables (each of which is assigned originally to hold all prefixes $P|w$ ($l_i \leq w < l_{i+1}$) under chosen DPL, as shown in Figure 1 by the leftmost component before collapsing) to yield high table utilization and is made set-associative to alleviate the overflow problem. Each entry in the LuHa table keeps a prefix pair for the two communicating networks, namely, sip (the source IP prefix) and dip (the destination IP prefix). While different (sip,

dip) pairs after being rounded down may become identical and distinct prefixes possibly yield the same hashed index, the set-associative degree of the LuHa table can be held low in practice. Given the LuHa table composed of 2^d sets, each with α entries, it experiences overflow if the number of rules hashed into the same set exceeds α . However, this overflow problem is alleviated, since a filter rule can be stored in either one of the two sets indexed by its sip and dip. With the LuHa table, our HaRP arrives at (1) rapid packet classification due to a reduced number of hash probes through a provision of parallel accesses to all entries in a LuHa set and also to a restricted scope of search (pointed to by the matched LuHa entry) in the second stage, and (2) a low SRAM requirement due to one single set-associated hash table (for better storage utilization).

Generalized HaRP

Given a filter rule with its sip or dip being $P|w$ and under $DPL = \{l_1, l_2, \dots, l_i, \dots, l_m\}$, HaRP can be generalized by rounding down $P|w$, with $l_i \leq w < l_{i+1}$, to $P|l_b$, for all $1 \leq b \leq i$, before hashing $P|l_b$ to identify more candidate sets for keeping the filter rule. In other words, this generalization in rounding down prefixes lets a filter rule be stored in any one of those 2^{i-1} sets hashed by $P|l_b$ in the LuHa table, referred to as $HaRP^*$. This is possible because HaRP takes advantage of the “*transitive property*” of prefixes – for a prefix $P|w$, $P|t$ is a prefix of $P|w$ for all $t < w$, considerably boosting its pseudo set-associative degree. A classification lookup for an arrived packet under DPL with m treads involves m hash probes via its source IP address and m probes via its destination IP address, therefore allowing the prefix pair of a filter rule (say, $(P_s|w_s, P_d|w_d)$, with $l_s^s \leq w_s < l_{s+1}^s$ and $l_d^d \leq w_d < l_{d+1}^d$) to be stored in **any one** of the i^s sets indexed by round-down P_s (i.e., $P_s|\{l_1, l_2, \dots, l_s^s\}$, if P_s is not a wildcard), or **any one** of the i^d sets indexed by round-down P_d (i.e., $P_d|\{l_1, l_2, \dots, l_d^d\}$, if P_d is not a wildcard). HaRP^{*} balances the prefix pairs among many candidate sets (each with α entries), making the LuHa table behave like an $(i^s + i^d) \times \alpha$ set-associative design under ideal conditions to enjoy high storage efficiency. Given DPL with 5 treads: $\{28, 24, 16, 12, 1\}$, for example, HaRP^{*} rounds down the prefix of $010010001111001 \times (w=15)$ to $010010001111 (\zeta=12)$ and 0 ($\zeta=1$) for hashing.

This potentially high pseudo set-associativity makes it possible for HaRP^{*} to choose a small number of treads (m). A small m lowers the number of hash probes per lookup accordingly, thus improving lookup performance. Adversely, as m drops, more rules can be mapped to a given set in the LuHa table, requiring m to be moderate practically, say 6 or so. Note that a shorter prefix (either P_s or P_d) leads to fewer candidate sets for storing a filter rule, but the number of filter rules with shorter prefixes is smaller, naturally curbing the likelihood of set overflow. Furthermore, HaRP^{*} enjoys virtually no overflow, as long as α is greater than 2, to be seen in the following analysis.

Our basic HaRP stated earlier is denoted by HaRP¹ (where $P|w$, with $l_i \leq w < l_{i+1}$, is rounded down to $P|l_i$). Rounding down

$P|w$ to both $P|l_i$ and $P|l_{i-1}$, dubbed HaRP², specifies up to four LuHa table sets for the filter rule. Clearly, HaRP² experiences overflow *only when $2 \times i$ sets in the LuHa table are all full*. The following analyzes the LuHa table in terms of its effectiveness and scalability, revealing that for a fixed, small α (say, 4), its overflow probability is negligible, provided that the ratio of the number of LuHa table entries to the number of filter rules is a constant, say ρ .

Effectiveness and Scalability of LuHa Table

From a theoretic analysis perspective, the probability distribution could be approximated by a Bernoulli process, assuming a uniform hash distribution for round-down prefixes. (As round-down prefixes for real filter datasets may not be hashed uniformly, we performed extensive evaluation of HaRP^{*} under publicly available 9 real-world datasets, with the results provided in Section 4.2.) The probability of hashing a round-down prefix $P|l_i$ randomly to a table with r sets equals $1/r$. Thus, the probability for k round-down prefixes, out of n samples (i.e., the filter dataset size), hashing to a given set is $\binom{n}{k} (1/r)^k (1 - 1/r)^{n-k}$. As each set has α entries, we get prob.(overflow | k round-down prefixes mapped to a set, for all $k > \alpha$) =

$$1 - \sum_{k=0}^{\alpha} \binom{n}{k} (1/r)^k (1 - 1/r)^{n-k}, \text{ with } r = (n \times \rho) / \alpha.$$

The above expression can be shown to give rise to almost identical results over any practical range of n , for given ρ and α . When $\rho = 1.5$ and $\alpha = 4$, for example, the overflow probability equals 0.1316 under $n = 500$, and it becomes 0.1322 under $n = 100,000$. Consequently, under a uniform hashing distribution of round-down prefixes, the set overflow probability of HaRP^{*} holds virtually unchanged as the filter dataset size grows, indicating good scalability of HaRP^{*} with respect to its LuHa table. We therefore provide in Figure 2, the probability of overflowing a set with $\alpha = 4$ entries versus ρ (called the dilation factor) for one filter dataset size (i.e., $n = 100,000$) only. As expected, the overflow probability dwindles as ρ rises (reflecting a larger table). For $\rho = 1.5$ (or 2), the probability of overflowing a typical 4-way set-associative table is 0.13 (or 0.05).

HaRP¹ achieves better LuHa table utilization, since it permits the use of either sip or dip for hashing, effectively yielding “*pseudo 8-way*” if sip and dip are not wildcards. It selects the less occupied set in the LuHa table from the two candidate sets hashed on the non-wild carded sip and dip. The overflowing probability of HaRP¹ can thus be approximated by the likelihood of both candidate LuHa table sets (indexed by sip and dip) being fully taken (i.e., each with 4 active entries). In practice, the probability results have to be conditioned by the percentage of filter rules with wild carded IP addresses. With a wild carded sip

(or dip), a filter rule cannot benefit from using either sip or dip for hashing (since a wild carded IP address is never used for hashing). The set overflowing probability results of HaRP¹ with wild carded IP address rates of 60% and 0% are depicted in Figure 2. They are interesting due to their representative characteristics of real filter datasets used in this study (as detailed in Section 4.1; the rates of filter rules with wild carded IP addresses for 9 datasets are listed with the right box). With a dilation factor $\rho = 1.5$, the overflowing probability of HaRP¹ drops to 1.7% (or 8.6%), for the wildcard rate of 0% (or 60%).

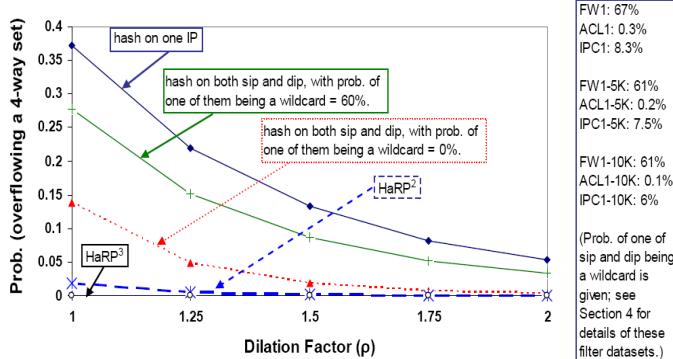


Figure 2. Overflow probability versus ρ for a 4-way set.

Meanwhile, HaRP² and HaRP³ are seen in the figure to outperform HaRP¹ smartly. In particular, HaRP² (or HaRP³) achieves the overflowing probability of 0.15% (or 1.4 E-07 %) for $\rho = 1.5$, whereas HaRP³ exhibits the overflowing probability less than 4.8 E-05 % even under $= 1.0$ (without any dilation for the LuHa table). These results confirm that HaRP^{*} indeed leads to virtually no overflow with $\alpha = 4$ under $* > 2$, thanks to its exploiting the high set-associative potential for effective table storage utilization. As will be shown in Section 4, HaRP^{*} also achieves great storage efficiency under real filter datasets, making it possible to hold a whole dataset in local cache practically for superior lookup performance.

Application-Specific Information (ASI) Table

The second stage of HaRP involves a table, each of whose entry keeps the values of application-specific filter fields (e.g., port numbers, protocol type) of one rule, dubbed the application-specific information (ASI) table (see Figure 1). If rules share the same IP prefix pair, their application-specific fields are stored in contiguous ASI entries packed as one chunk pointed by its corresponding entry in the LuHa table. For fast lookups and easy management, ASI entries are fragmented into chunks of a fixed size (say 8 contiguous entries). Upon creating a LuHa entry for one pair of sip and dip, a free ASI chunk is allocated and pointed to by the created LuHa entry. Any subsequent rule with an identical pair of sip and dip puts its application-specific fields in a

free entry insider the ASI chunk, if available; otherwise, another free ASI chunk is allocated for use, with a pointer established from the earlier chunk to this newly allocated chunk. In essence, the ASI table comprises linked chunks (of a fixed size), with one link for each (sip, dip) pair.

The number of entries in a chunk is made small practically (say, 8), so that all the entries in a chunk can be accessed simultaneously in one cycle, if they are put in one word line (of 1024 bits, which can physically comprise several SRAM modules). This is commonly achievable with current on-chip SRAM technologies. The ASI table requires a comparable number of entries as the filter dataset to attain desirable performance, with the longest ASI list containing 36 entries, according to our evaluation results based on real filter datasets outlined in Sections 4.3 and 4.4.

As demonstrated in Figure 1, each LuHa table entry is assumed to have 96 bits for accommodating a pair of sip and dip together with their 5-bit length indicators, a 16-bit pointer to an ASI list, and a 6-bit field specifying the ASI list length. Given the word line of 1024 bits and all entries of a set put within the same word line with on-chip SRAM technology for their simultaneous access in one cycle, the set-associative degree (α) of the LuHa table can easily reach 10 (despite that $\alpha = 4$ is found to be adequate in practice).

3.2 Installing Filter Rules

Given a set of filter rules, HaRP installs them by putting their corresponding field contents to the LuHa and the ASI tables sequentially. When adding a rule, one uses its source (or destination) IP prefix for finding a LuHa entry to hold its prefix pair after rounded down according to chosen DPL, if its destination (or source) IP field is a don't care (\times). Under HaRP^{*}, the number of round-down prefixes for a given non-wildcard IP prefix is up to $*$ (dependent upon the given IP prefix and chosen DPL). When both source and destination IP fields are specified, they are hashed separately (after rounded down) to locate an appropriate set for accommodation. The set is selected as follows: (1) if a hashed set contains the (sip, dip) prefix pair of the rule in one of its entry, the set is selected (and thus no new LuHa table entry is created to keep its (sip, dip) pair), (2) if none hashed set has an entry keeping such a prefix pair, a new entry is created to hold its (sip, dip) pair in the set with least occupancy; if all candidate sets are with the same occupancy, the last candidate set (i.e., the one indexed by the longest round-down dip) is chosen to accommodate the new entry created for keeping the rule. Note that a default table entry exists to hold the special pair of (\times, \times), and that entry has the lowest priority since every packet meets its rule.

The remaining fields of the rule are then put into an entry in the ASI table, indexed by the pointer stored in the selected LuHa entry. As ASI entries are grouped into chunks (with all entries inside a chunk accessed at the same time, in the way like accesses to those set entries in the LuHa table), the rule will find

any available entry in the indexed chunk for keeping the contents of its remaining fields, in addition to its full source and destination IP prefixes (without being rounded down). Should no entry be available in the indexed chunk, a new chunk is allocated for use (and this newly allocated chunk is linked to the earlier chunk, as described in Section 3.1).

```
Input: Received packet, with dip (destination IP address), sip, sport
       (source port), dport (destination port), proto (protocol type)

#define mask(L) ~((0x01 <<L) -1)
int match_rule_id = n_rules;

Hash_Probe (key_select) ::
key = (key_select == USE_DIP) ? dip : sip;
for each tread t in DPL {
    h = hash_func(key&mask(t), t); /* round down prefix & hash */
for each entry s in hash set LuHa[h] {
    if(PfxMatch((s.dip_prefix, dip), s.dip_prefix_length) &&
       PfxMatch((s.sip_prefix, sip), s.sip_prefix_length) {
        /* a prefix-pair matched, continue on checking ASI */
        for each asi entry e in the chunk pointed by s.asi_pointer {
            if(e.sport_low <= sport <= e.sport_high &&
               e.dport_low <= dport <= e.dport_high &&
               e.proto_low <= proto <= e.proto_high) {
                /* Match! Choose rule with lower rule number */
                if(match_rule_id >= e.ruleno)
                    match_rule_id = e.ruleno;
            }
        }
    }
}
/* Pass 1: hash via dip */
Hash_Probe(USE_DIP);
/* Pass 2: hash via sip */
Hash_Probe(USE_SIP);
```

Figure 3. Pseudo code for prefix-pair lookups.

3.3 Classification Lookups

Given the header of an incoming packet, a two-staged classification lookup takes place. During the LuHa table lookup, two types of hash probes are performed, one keyed with the source IP address (specified in the packet header) and the other with the destination IP address. Since rules are indexed to the LuHa table using the round-down prefixes during installation, the type of probes keyed by the source IP address involves m hash accesses, one associated with a length listed in $DPL = \{l_1, l_2, \dots, l_i, \dots, l_m\}$. Likewise, the type of probes keyed by the destination IP address also contains m hash accesses. This way ensures that no packet will be misclassified regardless of how a rule was installed, as illustrated by the pseudo code given in Figure 3.

Lookups in the ASI table are guided by the selected LuHa entries, which have pointers to the corresponding ASI chunks. The given source and destination IP addresses could match multiple entries (of different prefix lengths) in the LuHa table. Each matched entry points to one chunk in the ASI table, and the pointed chunks are all examined to find the best matched

rule. As all entries in one pointed chunk are fetched in a clock, they are compared concurrently with the contents of all relevant fields in the header of the arrival packet. If a match occurs to any entry, the rule associated with the entry is a candidate for application; otherwise, the next linked chunk is accessed for examination, until a match is found or the linked list is exhausted. When multiple candidate rules are identified, one with the longest matched (sip, dip) pair, or equivalently the lowest rule number, if rules are sorted accordingly, is adopted. On the other hand, if no match occurs, the default rule is chosen.

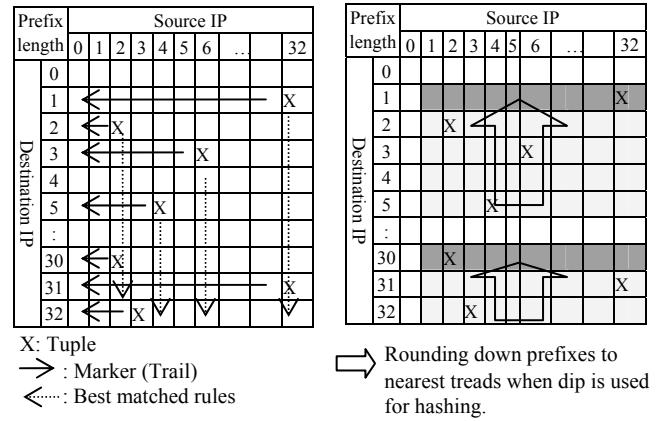


Figure 4. Comparison between TSS and proposed HaRP¹.

3.4 Lookup Time Complexity

Time complexity consists of search over both the LuHa table and the ASI table. Search over the LuHa table is indexed by keys composed of round-down prefix pairs (following the algorithm of Figure 3), taking exactly $2m$ hash probes under DPL with m treads (ranging from 4 to 8). On the other hand, search over the ASI table is directed by matched prefix pairs held in the LuHa table, and the mean number of such pairs is found to be smaller than 4 (for all nine filter datasets of sizes up to 10K rules adopted for our study, as listed in Table 1). Therefore, our HaRP requires 8-16 hash probes plus 4 ASI accesses per lookup, in comparison to 63 ($2W-1$) and $25 (\log^2 W)$, with W being the IP prefix length) probes respectively for Rectangle Search and Binary Tuple Search stated earlier. As a smaller m leads to fewer hash probes but more rules mapped to a given set in the LuHa table, selecting an appropriate m is important.

As explained in Section 2.2, TSS with optimization uses markers and pre-computed results to guide its search. However, the praised property (that any filter dataset usually comprises only a few unique prefix pair lengths) fails to take a role in optimization (which relies instead on each rule to leave markers), as depicted in Figure 4. Proliferating markers may heighten the storage requirement by an order of $O(N \times w)$. In contrast, HaRP based on DPL treads actually cuts the tuple space into segments along each dimension. When dip is used for hashing, as an example, all destination prefixes are rounded

down to designated length specified by the DPL set, as demonstrated in Figure 4 for HaRP¹ with designated prefix lengths equal to 30 and 1 shown. The selection of DPL can be made to match the distribution of unique prefix lengths for the best hashing results. Based on the fact that there are not many unique prefix pair length combinations [24, 25], HaRP design makes very efficient use of the LuHa table, in a way better than TSS over the tuple space. The storage requirement is a constant $O(N)$, linear to the number of rules.

3.5 Handling Incremental Rule Updates and Additional Fields

HaRP admits dynamic filter datasets very well. Adding one rule to the dataset may or may not cause any addition to the LuHa table, depending upon if its (sip, dip) pair has been present therein. An entry from the ASI table will be needed to hold the remaining fields of the rule. Conversely, a rule removal requires only to make its corresponding ASI entry available. If entries in the affected ASI chunk all become free after this removal, its associated entry in the LuHa table is released as well.

Packet classification often involves many fields, subject to large dimensionality. As the dimension increases, the search performance of a TSS-based approach tends to degrade quickly while needed storage may grow exponentially due to the combinatorial specification of many fields. By contrast, adding fields under HaRP does not affect the LuHa table at all, and they only need longer ASI entries to accommodate them, without increasing the number of ASI entries. Search performance hence holds unchanged in the presence of additional fields.

4 Evaluation and Results

This section evaluates HaRP using the publicly available filter databases, focusing on the distribution results of prefix pairs in the LuHa table. Because the LuHa table is consulted $2m$ times for DPL with m treads, the distribution of prefix pairs plays a critical role in hashing performance. Our evaluation assumes a 4-way set-associative LuHa table design, with default DPL comprising 8 treads: $\{32, 28, 24, 20, 16, 12, 8, 1\}$, chosen conveniently, not necessary to yield the best results. It will show that our use of a single set-associative table obtained by collapsing individual hash tables (see Figure 1) is effective.

This work assumes overflows to be handled by linked lists, and each element in the linked list contains 4 entries able to hold 4 additional prefix pairs. HaRP is compared with other algorithms, including the Tuple Space Search, BV, and HyperCuts in terms of the storage requirement and measured execution time on a multi-core SoC.

4.1 Filter Datasets

Our evaluation employed the filter database suite from the open source of ClassBench [26]. The suite contains three seed filter sets: covering Access Control List (ACL1), Firewall (FW1), and IP Chain (IPC1), made available by service

providers and network equipment vendors. By their different characteristics, various synthetic filter datasets with large numbers of rules are generated in order to study the scalability of classification mechanisms. For assistance in, and validation on, implementation of different classification approaches, the filter suite is accompanied with traces, which can also be used for performance evaluation as well [29]. The filter datasets utilized by our study are listed in the following table.

Table 1. Filter datasets

Seed Filters (#filters, trace length)	Synthetic Filters (#filters, trace length)	
ACL1(752, 8140)	ACL-5K(4415, 45600)	ACL-10K(9603, 97000)
FW1(269, 2830)	FW-5K(4653, 46700)	FW-10K(9311, 93250)
IPC1(1550, 17020)	IPC-5K(4460, 44790)	IPC-10K(9037, 90640)

4.2 Prefix Pair Distribution in LuHa Table

The hash function is basic to HaRP. In this article, a simple hash function is developed for use. First, a prefix key is rounded down to the nearest tread in DPL. Next, simple XOR operations are performed on the prefix key and the found tread length, as follows:

```
tread = find_tread_in_DPL(length of the prefix_key);
pfx = prefix_key & (0xffffffff << (32-tread)); // round down
h = (pfx) ^ (pfx>>7) ^ (pfx>>15) ^ tread ^ (tread<<5) ^
    (tread<<12) ^ ~(tread<<18) ^ ~(tread<<25);
set_num = (h ^ (h > 5) ^ (h << 13)) % num_of_set;
```

While better results may be achieved by using more sophisticated hash functions (such as cyclic redundancy codes, for example), it is beyond the scope of this article. Instead, we show that a single lumped LuHa table can be effective, and most importantly, HaRP^{*} works satisfactorily under a simple hash function.

The results of hashing prefix pairs into the LuHa table are shown in Figure 5, where the LuHa tables are properly sized. Specifically, the LuHa table is provisioned with $\rho = 2$ (*dilated* by a factor of 2 relative to the number of filter rules) for HaRP¹, whereas its size is then reduced by 25% (i.e., $\rho = 1.5$) to show how the single set-associative LuHa table performs with respect to fewer treads in DPL under HaRP^{*}. Figure 5(a) illustrates that HaRP¹ exhibits no more than 4% of overflowing sets in a 4-way set-associative LuHa table. Note that those results for 5K filter datasets (i.e., ACL-5K, FW-5K, and IPC-5K) were omitted in Figure 5 so that the remaining 6 curves can be read more easily, given that those omitted results lying between the set of results for 1K filter datasets and that for 10K datasets. Only the IPC1 dataset happens to have 20 prefix pairs mapped into one set. This congested set is caused partly by the non-ideal hash function and partly by the round-down mechanism of HaRP. Nevertheless, the single 4-way LuHa table exhibits good resilience in accommodating hash collisions for the vast majority (96%) cases.

When the number of DPL treads is reduced to 6 under HaRP^{*}, improved and well-balanced results can be observed in

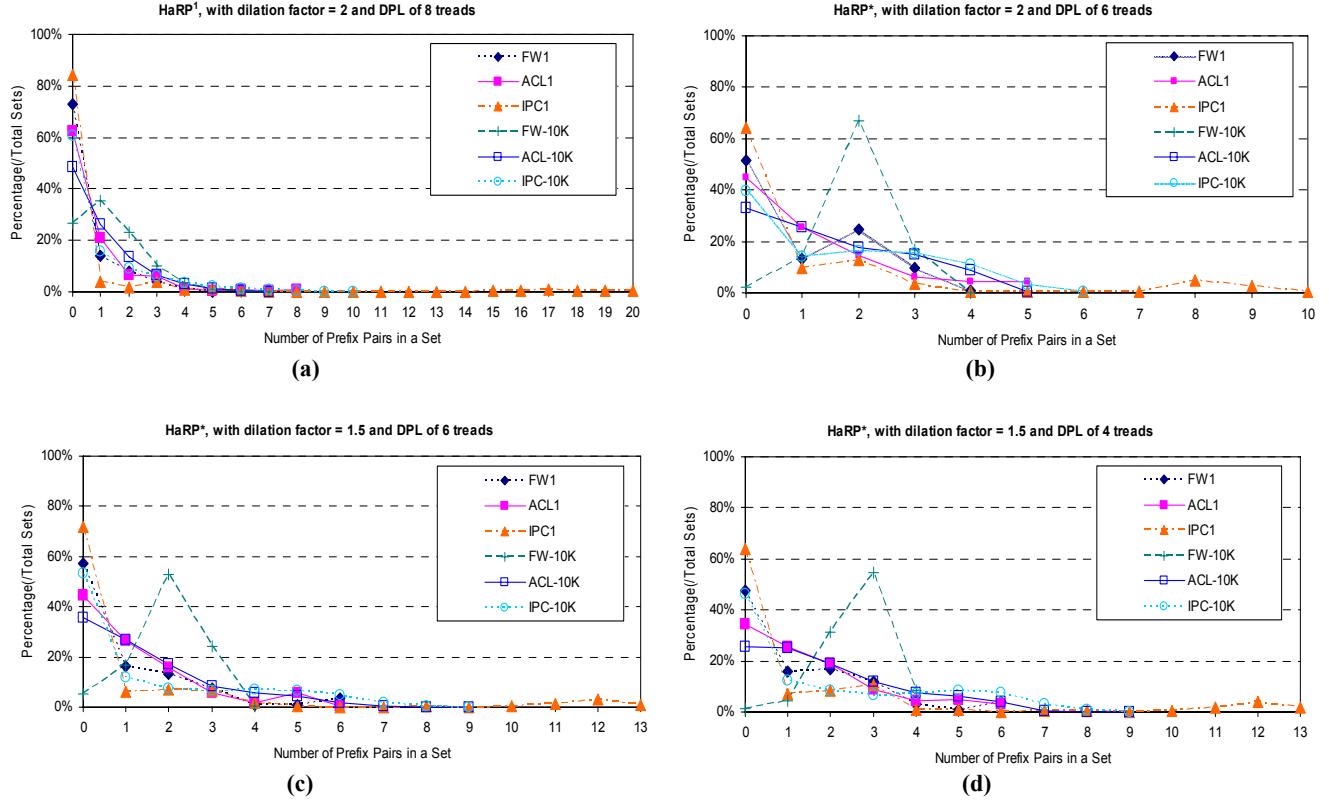


Figure 5. Results of hashing round-down prefixes into LuHa table.

Figure 5(b), where ρ equal 2. All datasets now experience less than 1% overflowing sets, except for ACL1 and IPC1 (which have some 4% and 8% overflows, respectively). Noticeably, even the most punishing case of IPC1 encountered in Figure 5(a) is reassured. These desirable results hold true when the LuHa table size is reduced by 25% and DPL contains fewer thread, as shown in Figures 5(c) and 5(d). Although a few congested sets emerge, they are still manageable. With 6 treads in DPL, fewer congested sets, albeit marginal, occur, as demonstrated in Figure 5(c), than with 4 threads depicted in Figure 5(d). This is expected, since the hash values are calculated over round-down prefixes, and a less number of treads leads to wider strides between consecutive treads, likely to make more prefixes identical in hash calculation after being rounded down. Furthermore, fewer treads in DPL implies a smaller number of LuHa table candidate sets among which prefix pairs can be stored. These results indicate that a single lumped set-associative table for HaRP* is promising in accommodating prefix pairs of filter rules in a classification dataset effectively.

4.3 Search over ASI Table

The second stage of HaRP probes the ASI (application-specific information) table, each of whose entry holds values of all remaining fields, as illustrated in Figure 1. As LuHa table

search has eliminated all rules whose source and destination IP prefixes do not match, pointing solely to those candidate ASI entries for further examination. It is important to find out how many candidate ASI entries exist for a given incoming packet, as they govern search complexity involved in the second stage.

As described in Section 3.1, we adopt a very simple design which puts rules with the same prefix pairs in an ASI chunk. While a more optimized design with smaller storage and higher lookup performance may be achieved by advanced techniques and data structures, we study the effectiveness of HaRP by using basic linear lists because of its simplicity.

The ASI lists are generally short, as shown in Figure 6, where the results for 5K filter datasets were omitted again for clarity. Over 95% of them have less than 5 ASI entries each, and hence, linear search is adequate. The ACL1 dataset is an exception, experiencing a long ASI list with 36 entries. By scrutinizing the outcome, we found that this case is caused by a large number of rules specified for a specific host pair, leading to a poor case since those rules for such host pairs fall in the same list. Furthermore, those rules have the form of $(0:\text{max_destination_port}, \times, \text{tcp})$, that is, a range is specified for the destination port, with the source port being wild carded and the protocol being TCP. Importantly, the destination port range $(0, dp_i)$ for Rule i is a sub-range of $(0, dp_{i+1})$ for Rule $i+1$. This is believed to

represent a situation where a number of applications at the target host rein accesses from a designated host. Nevertheless, fetching all ASI entries within one chunk at a time (achievable by placing them in the same word line) helps to address long ASI lists, if present (since one ASI chunk may easily accommodate 8 entries, each with 80 bits, as stated in the next subsection).

Note that the ASI distribution is orthogonal to the selection of DPL and to the LuHa table size. Filter rules are put in the same ASI list only if they have the same prefix pair combination.

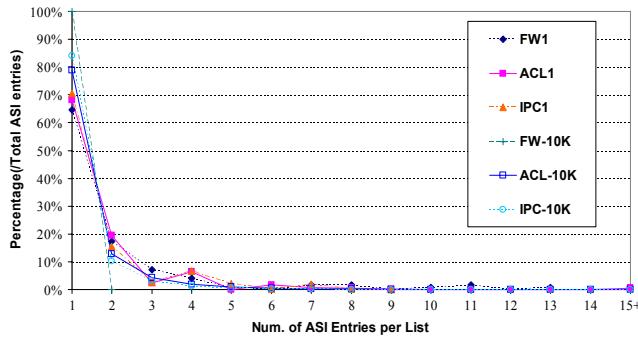


Figure 6. Length distribution of ASI link lists.

4.4 Storage Requirements

Table 2 shows memory storage measured for the rule datasets. Each LuHa entry is 12-byte long, comprising two 32b IP address prefixes, two 5b prefix length indicators, a 16b pointer to the ASI table, and a 6b integer indicating the length of its associated linked list. Each ASI entry needs 10 bytes to keep the port ranges and the protocol type, plus two bytes for the rule number (i.e., the priority).

Table 2. Memory size

	Total Storage (in KB, or otherwise MB as specified)				Per Rule Storage(Byte, or otherwise KB as specified)			
	HaRP	Tuple Space	BV	Hyper-Cuts	HaRP	Tuple Space	BV	Hyper-Cuts
FW1	4.64	22.72	10.50	10.19	17.66	86.49	40	36.79
ACL1	13.79	44.19	52.14	20.24	18.78	60.18	71	25.56
IPC1	29.17	56.26	92.33	91.19	19.27	37.17	61	58.25
FW-5K	101.0	629.5	3.07M	4.10M	22.23	138.5	691	922.3
ACL-5K	76.54	157.7	1.08M	136.8	17.75	36.57	257	29.73
IPC-5K	90.56	199.4	1.52M	332.6	20.79	45.79	358	74.34
FW-10K	217.3	1.68M	14.05M	25.05M	23.9	189.2	1.54K	2.75K
ACL-10K	192.5	403.4	7.31M	279.4	20.52	43.02	798	27.79
IPC-10K	187.5	449.8	6.79M	649.5	21.24	50.97	788	71.60

As listed in Table 2, HaRP enjoys clear superiority when compared with its previous counterparts, whose implemented source codes were available publicly [29] and

employed to gather their respective results included here. HaRP dramatically reduces memory storage needed and demonstrates consistent levels of storage requirement across all datasets examined. Previous techniques, especially those using decision-tree- or trie-based algorithms, exhibit rather unpredictable outcomes because the size of a trie largely depends on if datasets have comparable prefixes to enable trie contraction; otherwise, a trie can grow quickly toward full expansion. Among prior techniques, tuple space search (TSS) [24] and HyperCuts [20] show better results, although they still require more memory than HaRP. Those listed outcomes generally indicate what can be best achieved by the cited techniques. For TSS, as an instance, Tuple Pruning is implemented, but not pre-computed markers which increase storage requirement (see Section 2.2 and Figure 4 for details). For HyperCuts, its refinement options are all turned on, including *rule overlapping* and *rule pushing* for the most optimization results [20].

The results of memory efficiency, defined as the ratio between the total storage of constituent data structures (which include the provisioned but not occupied entries for the LuHa table in HaRP) and the minimal storage required to keep all filter rules (as in a linear array of rules), for various algorithms are listed in Table 3.

Table 3. Memory efficiency

	HaRP ($\rho = 2$)	HaRP ($\rho = 1.5$)	Tuple Space	BV	Hyper-Cuts
FW1	1.62	1.35	3.60	1.67	1.93
ACL1	1.58	1.31	2.51	2.96	1.38
IPC1	1.58	1.31	1.55	2.54	3.01
FW-5K	1.59	1.32	5.77	28.83	46.21
ACL-5K	1.58	1.31	1.52	10.69	1.59
IPC-5K	1.58	1.31	1.91	14.89	3.82
FW-10K	1.58	1.31	7.88	65.93	141.0
ACL-10K	1.58	1.31	1.79	33.26	1.49
IPC-10K	1.59	1.37	2.12	32.83	3.68

There are a number of interesting findings. First of all, HaRP consistently delivers greater efficiency than all other algorithms. When the LuHa table is diluted by a factor $\rho = 2$, all memory data structures allocated are no more than 50% of the amount required to keep the rules. If the LuHa table size is reduced to $\rho = 1.5$, total storage drops by 25%. In general, a smaller LuHa table yields lower performance because of more hash collisions. However, the next section will show measured results on multi-core systems under a small LuHa table (with $\rho = 1.5$) and small DPL to deliver satisfactory performance comparable to that under larger tables.

Contrary to HaRP enjoying consistent efficiency always, all other methods exhibit unsteady results. When the number of filter rules is small, those methods may achieve reasonable memory efficiency. As the dataset size grows, their efficiency results vary dramatically. For HyperCuts [20] (which uses a multi-way branch trie), its size largely depends on if datasets

have comparable prefixes that enable trie contraction; otherwise, the trie can grow exponentially toward full expansion. A decision tree-based method suffers from the fact that its number of kept rules may blow up quickly under a filter dataset with plentiful wild-carded rules. The less specific filter rules are, the lower memory efficiency it becomes, because a wild-carded rule holds true for all children at a node irrespective of the number of branches (cuts) made therein. (We have seen consistent trends for large datasets comprising 20K and 30K rules generated using the tool included in the ClassBench [26].) As analyzed in Section 3.1 and shown in Figure 2, the FW applications have over 60% wild-carded IP addresses (versus some 0.1% to 8% for ACL and IPC), yielding the worst memory efficiency consistently in Table 3. To a large degree, TSS [24] and BV [13] also leverage tries to narrow the search scope and hence are subject to the same problem. Furthermore, TSS employs one hash table per tuple in the space, likely to bloat the memory size because of underutilized hash tables. For BV, the n -bit vector stored at each leaf node of a trie is the main culprit for being memory guzzler.

Section 5.2 will demonstrate the measured performance results of HaRP, revealing that it not only achieves the best memory efficiency among all known methods but also classifies packet at four times faster than HyperCuts, and an order of magnitude higher than TSS and BV, under our multi-core evaluation platform.

5 Scalability and Lookup Performance on Multi-Cores

As each packet can be handled independently, packet classification suits a multi-core system well [6]. Given a multi-core processor with np cores, a simple implementation may assign a packet to any available core at a time so that np packets can be handled in parallel by np cores.

In this section, we present and discuss performance and scalability of HaRP in comparison with those of its counterparts BV [13], TSS [24], and HyperCuts [20]. Two HaRP configurations are considered: (1) basic HaRP with the LuHa table under a dilation factor $\rho = 2$ and with 8 treads in DPL, and (2) HaRP* with the LuHa table under $\rho = 1.5$ and with only 4 treads in DPL. By comparing results obtained for basic HaRP and HaRP*, we can gain insight into how the LuHa table size and the number of treads affect lookup performance.

For gathering measures of interest on our multi-core platform, our HaRP code was made multithreaded for execution. With those source codes for BV, TSS and HC implementations taken from the public source [29], we closely examined and polished them by removing unneeded data structures and also replacing some poor code segments with in order to get best performance levels of those referenced techniques. All those program codes were also made multithreaded to execute on the same multi-core platform, with their results presented in next sections.

5.1 Data Footprint Size

Because search is performed on each hashed set sequentially by a core, it is important to keep the footprint small so that the working data structure can fit into its caches, preferably the L1 (level-one) cache dedicated to a core. According to Table 3, HaRP requires the least amount of memory provisioned; Table 2 shows the actual data sizes to be much smaller. By our measurement, the FW-10K dataset has the largest size of some 200 KB. As a result, it is quite possible to hold the entire data structure in the L1 cache of a today’s core, even under large dataset sizes. This advantage in containing the growth of its data footprint size as the number of rules increases is unique to HaRP (and not shared by any prior technique), rendering it particularly suitable for multi-core implementation to attain high performance.

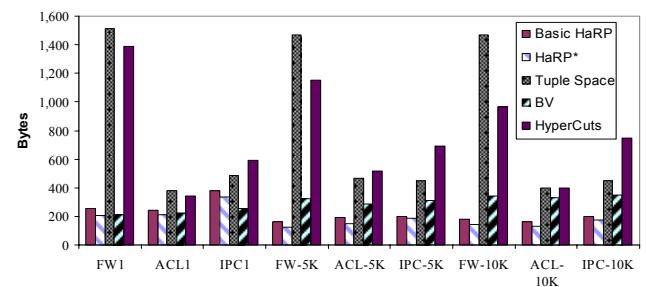


Figure 7. Average number of bytes fetched per lookup.

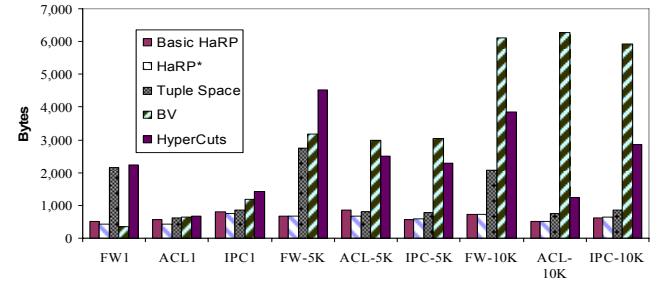


Figure 8. Worst case number of bytes accessed.

The behavior of HaRP driven by the traces provided with filter datasets [29] was evaluated to obtain the first order of measurement on the data footprint for lookups. Figure 7 depicts the mean number of bytes fetched per packet lookup, a conventionally adopted metric for comparing classification methods [20]. In general, HaRP enjoys lower average footprint per lookup, except when it is compared to BV under small filter datasets. Because HaRP always probes $2m$ LuHa sets (irrespective of the dataset size), it could incur more overhead than other techniques which use guided searches. However, when m is kept small and as the dataset size rises, our HaRP starts to prevail. Most importantly, as demonstrated in Figure 8, the deterministic procedure to probe $2m$ LuHa sets under m DPL treads yields more stable worst-case results across various rule datasets (which might possess different characteristics).

In the case of TSS, the data footprint is proportional to the

number of hash probes performed for a packet. In the firewall (FW) applications, TSS fetches 8 to 10 times more tuples (i.e., hash table accesses) than ACL and IPC applications, as depicted in the following table. As a result, the mean and the worst-case data footprints for FW are all far larger than those for ACL and IPC. In the next subsection, FW will be observed to deliver much lower classification rates due to its excessive hash probes.

Table 4. Mean number of accessed tuples per lookup (TSS)

FW1	ACL1	IPC1	FW-5K	ACL-5K	IPC-5K	FW-10K	ACL-10K	IPC-10K
72.95	6.30	11.45	68.2	10.68	9.24	67.76	6.73	8.69

For HyperCuts, the results also fluctuate, depending on the depth of the decision tree and the number of rules that are pushed up from the leaves and stored at the intermediate nodes. Pushing common rule subsets upward, the trie structure is an important technique for saving storage in HC [20]. The idea is to keep a common set of rules at the parent node if the rules hold true for all of its child nodes. In this way, rules can be associated with non-leaf nodes to save storage by avoiding replicas at the leaves. Adversely, this optimization heuristic requires inspection of rules kept at the non-leaf nodes while traversing the trie during lookups. Hence, it can lead to a large data footprint, as shown in Figure 7.

For BV, the worst case happens when it needs to check every single bit of the n -bit vector obtained by matching each individual field (for n rules). As a result, the worst-case number of BV grows consistently with the number of rules, and it is also the biggest worst-case footprint among all techniques examined.

Table 5. Search performance (in terms of mean number of entries) per lookup under basic HaRP and HaRP^{*}

	LuHa Search		ASI Search		
	$\rho = 2$, HaRP	$\rho = 1.5$, HaRP [*]	$\rho = 2$, HaRP	$\rho = 1.5$, HaRP [*]	
	Mean number of prefix pair		Mean number of entries		
	Checked	Matched	Checked	Matched	Checked
FW1	14.32	1.28	10.42	1.20	2.22
ACL1	25.67	1.52	21.81	1.53	1.85
IPC1	39.47	2.03	34.50	1.98	1.73
FW-5K	16.69	1.01	11.71	1.01	1.20
ACL-5K	18.31	1.17	12.88	1.22	3.38
IPC-5K	21.13	1.39	19.03	1.58	1.66
FW-10K	19.37	1.00	14.76	1.01	1.00
ACL-10K	17.57	1.14	13.53	1.13	1.64
IPC-10K	21.64	1.36	17.94	1.53	1.64
					1.69

As can be observed in Figures 7 and 8, HaRP^{*} often exhibits smaller footprints than basic HaRP. Although the LuHa table under HaRP^{*} (with $\rho = 1.5$) is 25% smaller than that under basic HaRP (with $\rho = 2$) and consequently the former has

a lot more well populated hash sets (see Figure 5(d)) than the latter (see Figure 5(a)), the use of 4 DPL treads in HaRP^{*} saves 8 hash probes per classification lookup, in comparison to basic HaRP (namely, 8 probes to more occupied sets versus 16 probes to less occupied sets). The mean numbers of matched entries under two HaRP configurations differ only a little, as depicted in Table 5, where the first and the third result columns list the average numbers of prefix pairs inspected per packet classification under basic HaRP and HaRP^{*}, respectively. Clearly, HaRP^{*} touches and inspects fewer prefix pairs than basic HaRP, due to fewer hash probes. The second and the fourth column contain the average numbers of prefix pairs matched. On average, less than two prefix pairs match in the LuHa table per classification lookup, signifying that the two-stage lookup procedure of HaRP is effective. Finally, the last two columns list the mean numbers of ASI tuples inspected with respect to each matched prefix pair. The mean numbers are small, suggesting that linear search as being performed in this work may suffice. Obviously, a more sophisticated scheme (such as a trie) could be employed, if ASI lists are long and sequential search becomes inefficient.

The next subsection presents measured execution time results when basic HaRP and HaRP^{*} are executed on our multicore platform, uncovered that HaRP^{*} outperforms its basic counterpart, because it incurs few hashing probes and accesses to more populated sets for better caching behavior.

5.2 Measured Performance on BCM-1480 MultiCore SoC

While data footprint results presented in the last subsection might reveal relative performance of different classification techniques (given the memory system is generally deemed as the bottleneck), computation steps or the mechanisms involved in dealing with the data structures are equally important and have to be taken into consideration. To arrive at more accurate evaluation, we executed all classification programs on a platform comprising a Broadcom's BCM-1480 4-core SoC [18]. BCM 1480 has four 700MHz SB-1™ MIPS cores [12], with each SB-1™ core a four-way in-order issue, superscalar design with separate 32K four-way set-associative instruction and data caches. The non-blocking data cache supports 8 outstanding misses. The cores are connected by a high-speed ZBbus and a unified 1MB, L2 cache keeps the active data structures to back up the smaller L1 caches. The memory system supports at most two x64 400MHz DDR channels, but our evaluation platform is equipped with only one channel clocked at 280MHz, giving rise to theoretical memory bandwidth of 35 Gbps.

Performance for HaRP, BV, TSS, and HC (HyperCuts) is measured. TSS generally holds its promise on a reduced number of hash probes it requires. In this implementation, two tries (one for source IP and another for destination IP) were constructed. During lookups, LPM (longest prefix matching) to the two tries produced two lists of candidate tuples, each realized by one hash table. Corresponding hash tables in the

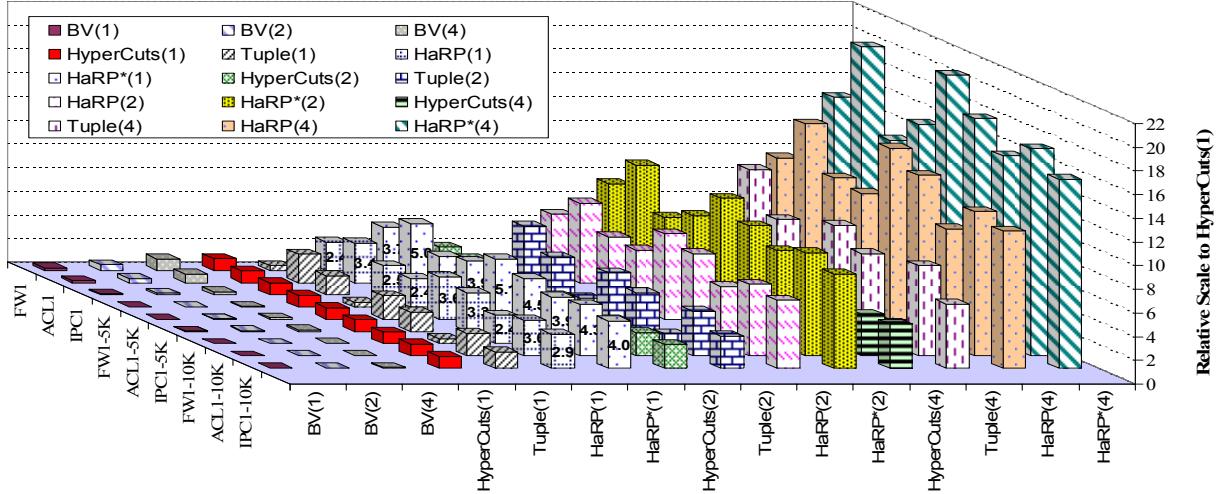


Figure 9. Measured throughput results on Broadcom BCM-1480 4-core SoC (in relative scale).

intersection of the two lists (namely, intersected tuples) are then probed. All executed programs were made multithreaded such that up to 4 threads could be launched to take advantage of the 4 SB-1™ cores. Millions of packets were generated from the traces packaged together with the rule datasets to measure the real elapsed times via the BCM-1480 ZBus counter, which ticks at every system clock.

Results depicted in Figure 9 are all relatively scaled to **one thread** HyperCuts performance, which is shown as a consistent scale of one across the graph for clear and system configuration-independent comparison. Labels on the x-axis of Figure 9 denote different techniques (i.e., BV, HyperCuts, TSS, and HaRP) executed on varying numbers of BCM-1480 cores (i.e., 1, 2, and 4). For example, BV(2) (or Tuple(4)) refers to BV (or TSS) run on 2 (or 4) cores. When the number of threads rises from 1 to 2 and then 4, HC shows a nearly linear scalability (in terms of raw classification rates) with respect to the number of cores. This scalability trend indeed exists for all techniques because packet classification is inherently parallel, as expected.

Overall, HaRP demonstrates the highest throughput among all techniques. On a per core basis, HaRP consistently delivers 2.4 to 3.5 times improvement over HC under the nine filter datasets. When compared with TSS, basic HaRP performs 2 to 3 times better than TSS under ACL and IPC filter datasets, and 8 times under the firewall applications (FWs). This is because HaRP requires fewer hash probes than TSS under firewall datasets. Our HaRP always performs $2m$ lookups, equal to 16 for $m = 8$. Contrary to HaRP, TSS performs as many as four times more hash probes under Firewall (see Table 4). For ACL and IPC datasets, TSS may require slightly fewer hash table lookups, but that advantage is more than negated by its two LPM search passes over the tries, with respect to the source and the destination IP prefixes. Furthermore, the smaller data footprint enjoyed by HaRP (demonstrated in Figure 7) leads to better cache performance.

Relative performance exhibited by HaRP* is even greater than that by basic HaRP, stemming from the fact it employs DPL with 4 treads, as opposed to 8 treads for HaRP. This brings the number of hash probes per lookup from 16 down to 8, incurring less hashing overhead. Most importantly, HaRP* is expected to be more caching-friendly, because accessing prefix pairs located in 8 sets should enjoy better caching locality than prefix pairs spread across 16 sets. Even though HaRP* uses a LuHa table which is 25% smaller than that of HaRP, HaRP* outperforms HC (or TSS) by 4 to 5 times (or 3 to 10 times), on an average, under the nine datasets, as demonstrated in Figure 9.

When compared to HC, BV shows poor performance with O(10) degradation, especially for large filter datasets. Because it starts with five LPM search processes across separate tries for individual header fields to produce a list of candidate rules in order to get a 5-field cross product, BV is inefficient for software implementation run on a multi-core platform, since its processor caches are expected to be trashed due to the large footprint incurred, as revealed in Figures 7 and 8. Thus, BV is better suitable for custom hardware with parallelism supported by high memory bandwidth, suffering from poor scalability.

Table 4 lists the average number of tuples (i.e., hash tables) fetched per packet lookup under TSS, with respect to different filter datasets examined. Hash probes for firewall applications (FWs) are far more than those for ACL and IPC datasets. This is consistent with the results of Figures 7 and 8, where FWs exhibit large footprints. Under FWs, TSS delivers 50% to 70% less performance than HC on a per-core basis. However, TSS outperforms HC under ACL and IPC datasets by as much as nearly 100%.

According to the average footprint results given in Figure 7, it does not seem that TSS can outperform HC in such a wide margin. For ACL-5K and ACL-10K datasets, HC reads roughly the same amount (but no more than 10%) of data bytes as TSS. However, TSS delivers almost 100% higher

throughputs per core. Under IPC-5K and IPC-10K, TSS fetches about 50% less data than HC and shows 47% higher throughput. It confirms that the data footprint can indeed give first-order estimation on how well a technique could perform, but the code path during execution is nevertheless critical. By inspecting the disassembled HC code, we found that the code path for HC could be long. For example, at each step traversing the decision tree, the number of bits to be extracted from a field needs to be determined, and next the extracted bits are used to calculate the location of the next child in the decision tree. In brief, the total number of splits (i.e., children) of a node is specified by $NC = \prod_i nc(i)$, where $nc(i)$ is the number of cuts performed on the i^{th} header field. During search, $\log_2(nc(i))$ bits are extracted from the appropriate positions in the i^{th} field; assuming the decimal value represented by the extracted bits is v_i , the number of child positions in the linear array covering the NC space is then expressed by $\sum_{i=1}^{D-1} v_i \times \prod_{j=i+1}^D nc(j) + v_D$ for D dimensions. These operations seem simple, but in fact, they can take hundreds of cycles to complete, causing a significant performance loss, as observed above.

6 Concluding Remarks

Packet classification is essential for most network system functionality and services, but it is complex since it involves comparing multiple fields in a packet header against entries in the filter dataset to decide the proper rule to apply for handling the packet [9]. This article has considered a rapid packet classification mechanism realized by hashing round-down prefixes (HaRP) able to not only exhibit high scalability in terms of both the classification time and the SRAM size involved, but also effectively handle incremental updates to the filter datasets. Based on a single set-associative LuHa hash table (obtained by lumping a set of hash table units together) to support two-staged search, HaRP promises to enjoy better classification performance than its known software-oriented counterpart, because the LuHa table narrows the search scope effectively based on the source and the destination IP addresses of an arrival packet during the first stage, leading to fast search in the second stage. With its required SRAM size lowered considerably, HaRP makes it possible to hold entire search data structures in the local cache of each core within a contemporary processor, further elevating its classification performance.

The LuHa table admits each filter rule in a set with lightest occupancy among all those indexed by $\text{hash}(\text{round-down sip})$ and $\text{hash}(\text{round-down dip})$, under HaRP*. This lowers substantially the likelihood of set overflow, which occurs only when all indexed sets are full, attaining high SRAM storage utilization. It also leads to great scalability, even for small LuHa table set-associativity (of 4), as long as the table is diluted by a small factor (say, $\rho = 1.5$ or 2). Our evaluation results have shown that HaRP* with the set associative degree of 4, generally experiences very rare set overflow instances (i.e., no more than

1% of those sets in the LuHa table with $\rho = 2$ under all studied filter datasets other than ACL1 and IPC1, if DPL has 6 treads).

Empirical assessment of HaRP has been conducted on our platform comprising a Broadcom's BCM-1480 SoC [18], which has four 700MHz SB-1™ MIPS cores [12]. A simple hashing function was employed for our HaRP implementation. Extensive measured results demonstrate that HaRP* outperforms HC [20] (or TSS [24]) by 4 to 5 times (or 3 to 10 times), on an average, under the nine databases examined, when its LuHa table is with $\rho = 1.5$ and there are 4 DPL treads. Besides its efficient support for incremental rule updates, our proposed HaRP also enjoys far better classification performance than previous software-based techniques.

Note that theoretically pathological cases may occur despite encouraging pragmatic results by HaRP*, as we have witnessed in this study. For example, a large number of (hosts on the same subnet with) prefixes $P|w$ can differ only in a few bits. Hence, those prefixes can be hashed into the same set after being rounded down, say $P|w$ to $P|l_i$, for $l_i \leq w < l_{i+1}$, under HaRP*. There are possible ways to deal with such cases and to avoid overwhelming the indexed set. A possible means is to use one and only one entry to keep the round-down prefix $P|l_i$, as opposed to holding all $P|w$'s in individual entries following the current design. Subsequently, the $(w - l_i)$ round-down bits can form a secondary indexing structure to provide the differentiation (among rules specific to each host) and/or the round-down bits can be mingled with the remaining fields of the filter rules. Thus, each stage narrows the range of search by small and manageable structures. These possible options will be explored in the future.

References

- [1] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *Proceedings of 20th Annual Joint Conf. of IEEE Computer and Communications Societies (INFOCOM 2001)*, pp. 1454–1463, Apr. 2001.
- [2] F. Chang *et al.*, "Efficient Packet Classification with Digest Caches," *Proceedings of Workshop on Network Processors and Applications (NP-3*, in conjunction with 10th Int'l Conference on High-Performance Computer Architecture), Feb. 2004.
- [3] W. T. Chen, S. B. Shih, and J. L. Chiang, "A Two-Stage Packet Classification Algorithm," *Proceedings of 17th International Conference on Advanced Information Networking and Applications (AINA '03)*, pp. 762–767, Mar. 2003.
- [4] Y. H. Cho and W. H. Magione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," *Proceedings of 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 125–134, Apr. 2004.
- [5] Y.-T. Chen and S.-S. Lee, "An Efficient Packet Classification Algorithm for Network Processors," *Proc. of IEEE Int'l Conf. on Communications (ICC 2003)*, pp. 1596–1600, May 2003.
- [6] H. Cheng *et al.*, "Scalable Packet Classification Using Interpreting a Cross-Platform Multi-Core Solution," *Proceedings 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel*

- Programming (PPoPP '08)*, pp. 33-42, Feb. 2008.
- [7] S. Dharmapurikar *et al.*, “Fast Packet Classification Using Bloom Filters,” *Proc. ACM/IEEE Symp. Architectures for Networking and Communications Systems (ANCS '06)*, pp. 61-70, Dec. 2006.
 - [8] Q. Dong *et al.*, “Wire Speed Packet Classification without TCAMs: A Few More Registers (and a Bit of Logic) Are Enough,” *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pp. 253-264, June 2007.
 - [9] P. Gupta and N. McKeown, “Packet Classification on Multiple Fields,” *Proceedings of ACM Annual Conference of Special Interest Group on Data Communication (SIGCOMM '99)*, pp. 147-160, Aug./Sept. 1999.
 - [10] P. Gupta and N. McKeown, “Classifying Packets with Hierarchical Intelligent Cuttings,” *IEEE Micro*, vol. 20, pp. 34-41, Jan. 2000.
 - [11] A. Kennedy, X. Wang, and B. Liu, “Energy Efficient Packet Classification Hardware Accelerator,” *Proceedings of IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pp. 1-8, Apr. 2008.
 - [12] D. Kruckemyer, “The SB-1™ Core: A High Performance, Low Power MIPS™ 64 Implementation,” *Proceedings of IEEE Symp. on High Performance Chips (Hot Chips 12)*, Aug. 2000.
 - [13] T. V. Lakshman and D. Stiliadis, “High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching,” *Proc. of ACM Annual Conference of Special Interest Group on Data Communication (SIGCOMM '98)*, pp. 191-202, Aug./Sept. 1998.
 - [14] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, “Algorithms for Advanced Packet Classification with Ternary CAMs,” *Proceedings of ACM Annual Conference of Special Interest Group on Data Communication (SIGCOMM 2005)*, pp. 193-204, Aug. 2005.
 - [15] F.-Y. Lee and S. Shieh, “Packet Classification Using Diagonal-Based Tuple Space Search,” *Computer Networks*, vol. 50, pp. 1406-1423, 2006.
 - [16] J. van Lunteren and T. Engbersen, “Fast and Scalable Packet Classification,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 560-571, May 2003.
 - [17] F. Pong and N.-F. Tzeng, “Storage-Efficient Architecture for Routing Tables via Prefix Transformation,” *Proc. 32nd IEEE Conf. on Local Computer Networks (LCN 2007)*, pp. 55-62, Oct. 2007.
 - [18] S. Santhanam *et al.*, “A 1GHz Power Efficient Single Chip Multiprocessor System for Broadband Networking Applications,” *Proc. of 15th Symp. on VLSI Circuits*, June 2001, pp. 107-110.
 - [19] D. Shah and P. Gupta, “Fast Incremental Updates on Ternary-CAMs for Routing Lookups and Packet Classification,” *Proc. of 8th Annual IEEE Symposium on High-Performance Interconnects (Hot Interconnects 8)*, pp. 145-153, Aug. 2000.
 - [20] S. Singh *et al.*, “Packet Classification using Multidimensional Cutting,” *Proceedings of ACM Annual Conference of Special Interest Group on Data Communication (SIGCOMM 2003)*, pp. 213-214, Aug. 2003.
 - [21] H. Song and J. W. Lockwood, “Efficient Packet Classification for Network Intrusion Detection Using FPGA,” *Proceedings of ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays (FPGA '05)*, pp. 238-245, Feb. 2005.
 - [22] E. Spitznagel, D. Taylor, and J. Turner, “Packet Classification Using Extended TCAMs,” *Proceedings of 11th IEEE Int'l Conf. on Network Protocols (ICNP '03)*, pp. 120-131, Nov. 2003.
 - [23] V. Srinivasan *et al.*, “Fast and Scalable Layer Four Switching,” *Proc. of ACM Annual Conference of Special Interest Group on Data Communication (SIGCOMM '98)*, pp. 191-202, Sept. 1998.
 - [24] V. Srinivasan, S. Suri, and G. Varghese, “Packet Classification Using Tuple Space Search,” *Proceedings of ACM Annual Conference of Special Interest Group on Data Communication (SIGCOMM '99)*, pp. 135-146, Aug./Sept. 1999.
 - [25] D. E. Taylor, “Survey and Taxonomy of Packet Classification Techniques,” *ACM Computing Surveys*, vol. 37, no. 3, pp. 238-275, Sept. 2005.
 - [26] D. E. Taylor and J. S. Turner, “ClassBench: A packet Classification Benchmark,” *Proc. 24th IEEE Int'l Conference on Computer Communications (INFOCOM 2005)*, March 2005.
 - [27] G. Wang and N.-F. Tzeng, “TCAM-Based Forwarding engine with Minimum Independent Prefix Set (MIPS) for Fast Updating,” *Proceedings of IEEE International Conference on Communications (ICC '06)*, June 2006.
 - [28] P. Warkhede, S. Suri, and G. Varghese, “Fast Packet Classification for Two-Dimensional Conflict-Free Filters,” *Proc. 20th Annual Joint Conf. of IEEE Computer and Communications Societies (INFOCOM 2001)*, pp. 1434-1443, Apr. 2001.
 - [29] Washington University, “Evaluation of Packet Classification Algorithms,” at <http://www.arl.wustl.edu/~hs1/PClassEval.html>.
 - [30] Z. Wu, M. Xie, and H. Wang, “Swift: A Fast Dynamic Packet Filter,” *Proceedings of 5th USENIX Networked Systems Design and Implementation (NSDI '08)*, pp. 279-292, Apr. 2008.

Tolerating File-System Mistakes with EnvyFS

Lakshmi N. Bairavasundaram[†], Swaminathan Sundararaman,

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Computer Sciences Department, University of Wisconsin-Madison

Abstract

We introduce *EnvyFS*, an N-version local file system designed to improve reliability in the face of file-system bugs. *EnvyFS*, implemented as a thin VFS-like layer near the top of the storage stack, replicates file-system metadata and data across existing and diverse commodity file systems (e.g., ext3, ReiserFS, JFS). It uses majority-consensus to operate correctly despite the sometimes faulty behavior of an underlying commodity *child* file system. Through experimentation, we show *EnvyFS* is robust to a wide range of failure scenarios, thus delivering on its promise of increased fault tolerance; however, performance and capacity overheads can be significant. To remedy this issue, we introduce *SubSIST*, a novel single-instance store designed to operate in an N-version environment. In the common case where all child file systems are working properly, *SubSIST* coalesces most blocks and thus greatly reduces time and space overheads. In the rare case where a child makes a mistake, *SubSIST* does not propagate the error to other children, and thus preserves the ability of *EnvyFS* to detect and recover from bugs that affect data reliability. Overall, *EnvyFS* and *SubSIST* combine to significantly improve reliability with only modest space and time overheads.

1 Introduction

File systems make mistakes. A modern file system consists of many tens of thousands of lines of complex code; such a system must handle memory-allocation failure, disk faults, and system crashes, and in all cases preserve the integrity of both user data and its own metadata. Thus, it is perhaps no surprise that many recent studies have uncovered hundreds of bugs in file systems [14, 18, 34, 49, 51].

Bugs manifest in numerous ways. In the best case, a system immediately crashes; recent research has shown how to cope with such “fail-stop” behavior by both isolating said file system from the rest of the kernel and transparently restarting it [16, 44]. However, in more

insidious scenarios, file-system bugs have been shown to accidentally corrupt the on-disk state of one or more blocks [34, 49, 51]; such “fail-silent” behavior is much more challenging to detect and recover from, and thus can lead to both data loss (due to a corrupt directory) or bad data passed back to the user.

One method to improve file systems and reduce fail-silent mistakes is thorough testing and other bug-finding techniques. For example, recent research has introduced a number of increasingly sophisticated and promising bug-finding tools [18, 29, 49, 51]. However, until such approaches are able to identify *all* file-system bugs, problems are likely to persist. Hence, file-system mistakes are here to stay; the challenge is how to cope with them.

In this paper, we advocate an approach based on the classic idea of N-version programming [1]. Specifically, we present the design and implementation of *EnvyFS*, a software layer that multiplexes file-system operations across multiple *child* file systems. *EnvyFS* issues all user operations to each child, determines the majority result, and delivers it to the user. By design, we thus eliminate the reliance on a single complex file system, instead placing it on a much simpler and smaller software layer.

A significant challenge in N-version systems is to formulate the common specification and to create the different versions. *EnvyFS* overcomes this challenge by using the Virtual File System (VFS) layer as the common specification and by leveraging existing Linux file systems already written by different open-source development groups (e.g., ext3 [46], JFS [8], ReiserFS [36]). In this manner, we build on work that leverages existing software bases to build N-version services, including NFS servers [37] and transaction-processing systems [47].

An important design goal in building *EnvyFS* is to keep it simple, thereby reducing the likelihood of bugs that arise from the sheer complexity of file-system code. At the same time, *EnvyFS* should leverage the VFS layer and existing file systems to the extent possible. We find that *EnvyFS* is indeed simple, being only a fraction of the

size as its child file systems, and can leverage much of the common specification. However, limitations do arise from the nature of the specification in combination with our goal of simplicity. For example, because child file systems issue different inode numbers for files, EnvyFS is tasked with issuing inode numbers as well; in the interest of simplicity, EnvyFS does not maintain these inode numbers persistently (i.e., the inode number for a file is the same within, but not across, mounts).

A second challenge for EnvyFS is to minimize the performance and disk-space overheads of storing and retrieving data from its underlying child file systems. Our solution is to develop a variant of a single-instance store (an SIS) [11, 17, 35]. By utilizing content hashes to detect duplicate data, an SIS can significantly reduce the space and performance overheads introduced by EnvyFS. However, using an SIS underneath EnvyFS mandates a different approach, as we wish to reduce overhead without sacrificing the ability to tolerate mistakes in a child file system. We achieve this by implementing a novel SIS (which we call SubSIST) that ensures that a mistake in one file system (e.g., filling a block with the wrong contents) does not propagate to other children, and thus preserves the ability of EnvyFS to detect faults in an underlying file system through voting. Thus, in the common case where all file systems work properly, SubSIST coalesces most blocks and can greatly reduce time and space overheads; in the rare case where a single child makes a mistake, SubSIST does not do so, enabling EnvyFS to detect and recover from the problem.

We have implemented EnvyFS and SubSIST for Linux; currently, EnvyFS employs any combination of ext3, JFS, and ReiserFS as child file systems. Through fault injection, we have analyzed the reliability of EnvyFS and have found that it can recover from a range of faults in nearly all scenarios; many of these faults cause irreparable data loss or unmountable file systems in the affected child. We have also analyzed the performance and space overheads of EnvyFS both with and without SubSIST. We have found across a range of workloads that, in tandem, they usually incur modest performance overheads. However, since our current implementation of SubSIST does not persist its data structures, the performance improvements achieved through SubSIST represent the best case. We find that SubSIST also reduces the space overheads of EnvyFS significantly by coalescing all data blocks. Finally, we have discovered that EnvyFS may also be a useful diagnostic tool for file-system developers; in particular, it helped us to readily identify and fix a bug in a child file system.

The rest of the paper is organized as follows. In Section 2, we present extended motivation. We present the design and implementation of EnvyFS and SubSIST in

Sections 3 and 4 respectively. We evaluate our system for reliability in Section 5 and performance in Section 6. We then discuss related work in Section 7 and conclude in Section 8.

2 Do File Systems Make Mistakes?

Before describing EnvyFS, we first briefly explain why we believe file systems do indeed make mistakes, and why those mistakes lead file systems to deliver corrupt data to users or corrupt metadata to themselves. Such failures are silent, and thus challenging to detect.

Recent work in analyzing file systems has uncovered numerous file system bugs, many of which lead to silent data corruption. For example, Prabhakaran et al. found that a single transient disk error could cause a file system to return corrupt data to the calling application [33, 34]. Further, a single transient write failure could corrupt an arbitrary block of the file system, due to weaknesses in the failure-handling machinery of the journaling layer [34]. Similar bugs have been discovered by others [50, 51].

Another piece of evidence that file systems corrupt their own data structures is the continued presence of file system check-and-repair tools such as fsck [30]. Despite the fact that modern file systems either use journaling [21] or copy-on-write [12, 19, 25, 38] to ensure consistent update of on-disk structures, virtually all modern file systems ship with a tool to find and correct inconsistencies in on-disk data structures [20]. One might think inconsistencies arise solely from faulty disks [6, 7]; however, even systems that contain sophisticated machinery to detect and recover from disk faults ship with repair tools [24]. Thus, even if one engineers a reliable storage system, on-disk structures can still become corrupt.

In addition to bugs, file systems may accidentally corrupt their on-disk structures due to faulty memory chips [31, 39]. For example, if a bit is flipped while a block is waiting to be written out, either metadata or data will become silently corrupted when the block is finally written to disk.

Thus, both due to poor implementations as well as bad memory, file systems can corrupt their on-disk state. The type of protection an N-version system provides is thus complementary to the machinery of checksums and parity and mirroring that could be provided in the storage system [28, 41], because these problems occur *before* such protection can be enacted. These problems cannot be handled via file-system backups either; backups potentially provide a way to recover data, but they do not help detect that currently-available data is corrupt. To detect (and perhaps recover) from these problems, something more is required.

3 EnvyFS: An N-Version File System

N-version programming [1, 2, 4, 5, 13, 15, 48] is used to build reliable systems that can tolerate software bugs. A system based on N-version programming uses N different versions of the same software and determines a majority result. The different versions of the software are created by N different developers or development teams for the same software specification. It is assumed (and encouraged using the specification) that different developers will design and implement the specification differently, lowering the chances that the versions will contain the same bugs or will fail in a similar fashion.

Developing N-version systems has three important steps (a) producing the specification for the software, (b) implementing the N different versions of the software, and (c) creating the environment that executes the different versions and determines a consensus result [1].

We believe the use of N-version programming is particularly attractive for building reliable file systems since the design and development effort required for the first two steps (i.e., specification and version development) can be much lower than for the typical case.

First, many existing commodity file systems adhere to a common interface. All Linux file systems adhere to the POSIX interface, which internally translates to the Virtual File System (VFS) interface. Thus, if an N-version file system is able to leverage the POSIX/VFS interface, then no additional effort will be needed to develop a new common specification. However, because the POSIX/VFS interface was not designed with N-versioning in mind, we do find that EnvyFS must account for differences between file systems.

Second, many diverse file systems are available for Linux today. For example, in Linux 2.6, there are at least 30 different file systems (depending upon how one counts), such as ext2, ext3, JFS, ReiserFS, XFS, FAT, and HFS; new ones are being implemented as well, such as btrfs. All have been built for the POSIX/VFS interface. These different file systems have drastically different data structures, both on disk and in memory, which reduces the chances of common file-system bugs. Furthermore, previous research has shown that file systems behave differently when they encounter partial-disk failures; for example, Prabhakaran et al. show that when directory data is corrupted, ReiserFS and JFS detect the problem while ext3 does not [34].

3.1 Design Goals and Assumptions

The design of EnvyFS is influenced by the following goals and assumptions:

Simplicity: As systems have shown time and again, complexity is the source of many bugs. Therefore, an N-version file system should be as simple as possible. In

EnvyFS, this goal primarily translates to avoiding persistent metadata; this simplification allows us to not allocate disk blocks and to not worry about failures affecting EnvyFS metadata.

No application modifications: Applications should not need to be modified to use EnvyFS instead of a single local file system. This goal supports our decision to leverage the POSIX specification as our specification.

Single disk: The N-version file system is intended to improve the reliability of desktop systems in the face of file-system mistakes. Therefore, it replicates data across multiple local file systems that use the same disk drive. This goal translates to a need for reducing disk-space overheads; thus, we develop a new single-instance store (Section 4) for our environment.

Non-malicious file systems: We assume that child file systems are not malicious. Thus, we must only guard against accidents and not intentional attempts to corrupt user data or file-system metadata.

Bug isolation: We also assume that the bugs do not propagate to the rest of the kernel. If such corruption were indeed a major issue, one could apply isolation techniques as found in previous work to contain them [16, 44].

3.2 Basic Architecture

EnvyFS receives application file operations, issues the operations to multiple *child file systems*, compares the results of the operation on all file systems, and returns the majority result to the application. Each child stores its data and metadata in its own disk partition.

We have built EnvyFS within Linux 2.6, and Figure 1 shows the basic architecture. EnvyFS consists of a software layer that operates underneath the virtual file system (VFS) layer. This layer executes file operations that it receives on multiple children. We use ext3 [46], JFS [9], and ReiserFS [36] for this purpose. We chose these file systems due to their popularity and their differences in how they handle failures [34]. However, the EnvyFS design does not preclude the use of other file systems that use the VFS interface.

Similar to stackable file systems [22], EnvyFS interposes transparently on file operations; it acts as a normal file system to the VFS layer and as the VFS layer to the children. It thus presents file-system data structures and interfaces that the VFS layer operates with and in turn manages the data structures of the child file systems. We have implemented wrappers for nearly all file and directory operations. These wrappers verify the status of necessary objects in the children before issuing the operation to them. For example, for an unlink operation, EnvyFS first verifies that both the file and its parent directory are consistent with majority opinion.

Each operation is issued in series to the child file systems; issuing an operation in parallel to all file systems

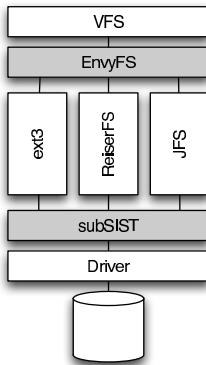


Figure 1: N-version file system in Linux. The figure presents the architecture of a 3-version file system with *ext3*, *ReiserFS* and *JFS* as the children. The core layer is *EnvysFS*; it is responsible for issuing file operations to all three file systems, determining a majority result from the ones returned by the file systems, and returning it to the *VFS* layer. The optional layer beneath the file systems (*SubSIST*) is a single-instance store built to work in an *N*-version setting; it coalesces user data stored by the different file systems in order to reduce performance and space overheads.

increases complexity and is unlikely to realize much, if any, performance benefit when the children share the same disk drive. When the operations complete, the results are semantically compared to determine the majority result; this result is then returned to the user. When no majority result is obtained, an I/O error is returned.

Our current implementation does not support the *mmap* operation. While supporting *mmap* is not fundamentally difficult, it does present a case where child file systems cannot be trivially leveraged. Specifically, an implementation of *mmap* in *EnvysFS* would likely involve the use of *file read* and *write* operations of children, rather than their *mmap* operations.

We now discuss how our implementation addresses each of the three steps of *N*-version programming. In particular, we discuss how *EnvysFS* deals with the complexities of leveraging the existing *POSIX* specification/*VFS* layer and of using existing Linux file systems while keeping *EnvysFS* simple.

3.3 Leveraging the *POSIX* Specification

EnvysFS leverages the existing *POSIX* specification and operates underneath *VFS* as it provides core functionality (like ordering of file operations) that is challenging to replicate without modifying applications. Thus, *EnvysFS* relies upon the correct operation of the *VFS* layer. We believe the *VFS* layer has been heavily tested over the years and is likely to have fewer bugs than the file systems themselves; this optimism is partially validated by Yang et al., who find two bugs in the *VFS* layer and nearly thirty in *ext3*, *ReiserFS*, and *JFS* [51].

One issue that *EnvysFS* must handle is that the *POSIX* specification is imprecise for use in an *N*-version setting; that is, the child file systems we leverage differ in various user-visible aspects that are not a part of the *POSIX* interface. For example, *POSIX* does not specify the order in which directory entries are to be returned when a directory is read; thus, different children may return directory entries in a different order. As another example, the inode number of a file is available to users and applications through the *stat* system call; yet, different file systems issue different inode numbers for the same file.

One approach to addressing this problem would be to make the specification more precise and change the file systems to adhere to the new specification. This approach has a number of problems. First, refining the specification discourages diversity across the different file systems. For example, if the specification details how inode numbers should be assigned to files, then all file systems will be forced to use the same algorithm to allocate inode numbers, perhaps causing them to also use the same data structures and inviting common bugs. Second, even given a more precise specification, non-determinism and differences in operation ordering can easily cause different results. Finally, changing the specification would greatly increase the amount of development effort to produce an *N*-version file system, since each existing Linux file system would need to be changed to use it as a child file system.

3.3.1 Semantic Result Comparison

Our solution is to have *EnvysFS* deal with the imprecise *POSIX* specification: when *EnvysFS* compares and returns results from the child file systems, it does so using semantic knowledge of how the *POSIX/VFS* interface operates. In other words, *EnvysFS* examines the *VFS* data structures returned by each child file system and does a semantic comparison of individual fields.

For example, for a file read operation, *EnvysFS* compares (a) the size of data read (or the error code returned), (b) the actual content read, and (c) the file position at the end of the read. For all file operations where inodes may be updated, *EnvysFS* compares (and copies to its *VFS* inode) the contents of the individual inodes. We have developed comparators for different file-system data types like superblocks, inodes, and directories. For example, an inode comparator checks whether the fields *i_nlink*, *i_mode*, *i_uid*, and so forth in the child inodes are the same. While *EnvysFS* compares results returned to it, it does not verify that the operation completed correctly in each file system; for example, it does not re-read data written to a file to verify that all file systems actually wrote the correct data.

As mentioned above, directory entries and inodes are especially interesting cases. We now describe how

EnvyFS handles these structures in more detail and we also describe how EnvyFS optimizes its comparison of data blocks across file systems.

Directory Entries: POSIX does not specify the order in which directory entries are to be returned. Thus, EnvyFS reads all directory entries from all file systems; it then returns individual entries that occur in a majority of file systems. The disadvantage of this approach is that it increases the overhead for the `getdirentries` system call for very large directories. We note that we could optimize the performance of this case (at the expense of code simplicity), by reading from child file systems only until EnvyFS finds matches for exactly as many entries as the user provides space for.

Inode Numbers: POSIX does not specify how inode numbers should be assigned to files, yet inode numbers are visible to user applications. Since EnvyFS cannot always use the inode number produced by any one child file system (because it may fail), it assigns a virtual inode number when a new object is encountered and tracks this mapping. Keeping with our simplicity goal, inode numbers so assigned are not persistent; that is, an object has a specific virtual inode number only between a mount and the corresponding unmount. This decision impacts only a few applications that depend on the persistence of file-system inode numbers. If applications using EnvyFS do require persistent inode numbers, one simple solution that could be explored is to store the inode mapping in a hidden file in the root directory of each file system and load the mapping at mount time. A specific example in this context is an NFS server using protocol versions 2 or 3; the server uses persistent inode numbers to create file handles for clients that can be used across server crashes. Even so, in protocol version 4, a “volatile file handle” option was introduced, thereby eliminating the need for persistent inode numbers. Interestingly, some local file systems, like the High Sierra file system for CD-ROMs, do not have persistent inode numbers [32].

Reads of Data Blocks: In performing read operations, we would like to avoid the performance overhead of allocating memory to store the results returned by all of the file systems (especially when the data read is already in cache). Therefore, EnvyFS reuses the memory provided by the application for the `read` system call. Reusing the memory influences two subsequent decisions. First, to determine whether the child file systems return the same data from the `read`, EnvyFS computes checksums on the data returned by the child file systems and compares them; a more thorough byte-by-byte comparison would require memory for all copies of data. Second, EnvyFS issues the `read` operation in series to child file systems only until a majority opinion is reached (i.e., usually to two children); this choice eliminates the problem of issuing reads again in case the last file system returns in-

correct data; in addition, in the common case, when file systems agree, the third read is avoided. It is important to note that we choose not to take the same issue-only-until-majority approach with other VFS operations such as `lookup` since the limited performance gain for such operations is not worth the complexity involved, say in tracking and issuing a sequence of lookups for the entire path when a `lookup` returns erroneous results in one file system. A future implementation could include a “verify-all” option that causes EnvyFS to issue the `read` to all file systems ignoring the performance cost.

In choosing the checksum algorithm for comparing data, one must remember that the cost of checksumming can be significant for reads that are satisfied from the page cache. We have measured that this cost is especially high for cryptographic checksums such as MD5 and SHA-1; therefore, in keeping with our goal of protecting against bugs but not maliciousness, we use a simple TCP-like checksum (sum of bytes) for comparisons.

3.3.2 Operation Ordering

Our placement of EnvyFS beneath VFS simplifies the issue of ordering file operations. As in many replication-based fault tolerance schemes, determining an ordering of operations is extremely important; in fact, recent work in managing heterogeneous database replicas focuses primarily on operation ordering [47]. In the context of a file system, consider the scenario where multiple file operations are issued for the same object: if an ordering is not predetermined for these operations, their execution may be interleaved such that the different children perform the operations in a different order and therefore produce different results even in the absence of bugs.

Unlike databases, the dependence between operations can be predetermined for file systems. In EnvyFS, we rely on the locking provided by the Linux VFS layer to order metadata operations. As explained earlier, this reliance cannot be avoided without modifying applications (to issue operations to multiple replicas of VFS that execute an agreement algorithm). In addition to the VFS-level locking, we perform file locking within EnvyFS for reads and writes to the same file. This locking is necessary since the VFS layer does not (and has no need to) order file reads and writes.

3.4 Using Existing File Systems

Our decision to leverage existing Linux file systems for child file systems greatly simplifies the development costs of the system. However, it does restrict our behavior in some cases.

One problem with using multiple local file systems is that the different file systems execute within the same address space. This exposes EnvyFS to two problems: (a) a kernel panic induced by a child file system, and (b) a memory bug in a child file system that corrupts the rest

of the kernel. A solution to both problems would be to completely isolate the children using a technique such as Nooks [43]. However, due to the numerous interactions between the VFS layer and the file systems, such isolation comes at a high performance cost.

Therefore, we explore a more limited solution to handle kernel panics. We find the current practice of file systems issuing a call to `panic` whenever they encounter errors to be too drastic, and developers seem to agree. For example, ext3 code had the following comment: *“Given ourselves just enough room to cope with inodes in which i_blocks is corrupt: we’ve seen disk corruptions in the past which resulted in random data in an inode which looked enough like a regular file for ext3 to try to delete it. Things will go a bit crazy if that happens, but at least we should try not to panic the whole kernel”*. In the case of ext3 and JFS, a mount option (`errors`) can specify the action to take when a problem is encountered; one could specify `errors=continue` to ensure that `panic` is not called by the file systems. However, this option is not available on all file systems. Our solution is to replace calls to `panic`, `BUG`, and `BUG_ON` by child file systems with a call to a `nvfs_child_panic` routine in EnvyFS. This simple replacement is performed in file-system source code. The `nvfs_child_panic` routine disables issuing of further file operations to the failed file system.

Another limitation of using existing file systems is that different file systems use different error codes for the same underlying problems (e.g., “Input/output error”, “Permission denied”, or “Read-only file system”). A consistent error code representing each scenario would enable EnvyFS to take further action. In our current implementation EnvyFS simply reports the majority error code or reports an I/O error if there is no majority.

3.5 Keeping EnvyFS Simple

EnvyFS has its own data structures (e.g., in-memory inodes and *dentry* structures), which are required for interacting with the VFS layer. In turn, EnvyFS manages the allocation and deallocation of such structures for child file systems; this management includes tracking the status of each object: whether it matches with the majority and whether it needs to be deallocated.

In keeping with our simplicity goal, we have designed EnvyFS so that it does not maintain any persistent data structures of its own. This decision affects various parts of the design; we previously discussed how this impacts the management of inode numbers (Section 3.3.1); we now discuss how it impacts the handling of faulty file systems and system crashes.

3.5.1 Handling Disagreement

An important part of EnvyFS is the handling of cases where a child file system disagrees with the majority re-

sult. This part is specifically important for local file systems since the ability to perform successive operations may depend on the result of the current operation (e.g., a file read cannot be issued when `open` fails).

When an error is detected, in order to restore EnvyFS to full replication, the erroneous child file system should be repaired. The repair functionality within EnvyFS fixes incorrect data blocks and inodes in child file systems. Specifically, if EnvyFS observes that the file contents in one file system differs from the other file systems during a file read, it issues a write of the correct data to the corrupt file system before returning the data to the user. With respect to inodes, EnvyFS repairs a subset of various possible corruptions; it fixes inconsistencies in the permission flags (which are `i_mode`, `i_uid`, `i_gid`) with the majority result from other file systems. It also fixes size mismatches where the correct size is larger than the corrupt one by copying the data from correct file systems. On the other hand, issuing a file truncate for the case where the correct size is smaller may result in more corruption in an already corrupt file system (e.g., the blocks being freed by truncate may actually be in use by a different file as a result of a prior corruption).

As the above example demonstrates, efficient repair for all inconsistencies is challenging. If EnvyFS cannot repair the erroneous object in a child file system, it operates in *degraded-mode* for the associated object. In degraded mode, future operations are not performed for that object in the file system with the error, but EnvyFS continues to perform operations on other objects for that file system. For example, if a child’s file inode is declared faulty, then read operations for that file are not issued to that file system. As another example, if a lookup operation completes successfully for only one file system, its corresponding in-memory *dentry* data structure is deallocated, and any future file create operation for that *dentry* is not issued to that file system.

For simplicity, the validity information for objects is not maintained persistently. With this approach, after a reboot, the child file system will try to operate on the faulty objects again. If the object is faulty due to a permanent failure, then the error is likely to be detected again, as desired. Alternately, if the problem was due to a transient error, the child will return to normal operation as long as the object has not been modified in the interim. Our current approach to fully repair inconsistencies that cannot be repaired in-flight requires that the entire erroneous child file system be re-created from the other (correct) children, an expensive process.

Some further challenges with efficient repair may arise from limitations of the VFS layer. Consider the following scenario. A file with two hard links to it may have incorrect contents. If EnvyFS detects the corruption through one of the links, it may create a new file in

the file system to replace the erroneous one. However, there is no simple way to identify the directory where the other link is located, so that it can be fixed as well (except through an expensive scan of the entire file system). In the future, we plan to investigate how one can provide hooks into the file system to enable fast repair.

3.5.2 System Crashes

When a system crash occurs, EnvyFS file-system recovery consists of performing file-system recovery for all child file systems before EnvyFS is mounted again. In our current approach, EnvyFS simply leverages the recovery methods inherent to each individual file system, such as replaying the journal. This approach leads to a consistent state within each of the children, but it is possible for different file systems to recover to different states. Specifically, when a crash occurs in the middle of a file operation, EnvyFS could have issued (and completed) the operation for only a subset of the file systems, thereby causing children to recover to different states. In addition, file systems like ext3 maintain their journal in memory, flushing the blocks to disk periodically; journaling thus provides consistency and not durability.

An alternative approach for solving this problem would be for EnvyFS itself to journal operations and replay them during recovery. However, this would require EnvyFS to maintain persistent state.

In EnvyFS, the state modifications that occur durably for a majority of file systems before the crash are considered to have completed. The differences in the minority set can be detected when the corresponding objects are read, either during user file operations or during a proactive file-system scan. There are corner cases where a majority result will not be obtained when a system crash occurs. In these cases, choosing the result of any one file system will not affect file-system semantics. At the same time, these cases cannot be distinguished from other real file-system errors. Therefore, EnvyFS returns an error code when these differences are detected; future implementations could choose to use the result from a designated “primary” child.

4 SubSIST: A Single-Instance Store

Two issues that arise in using an N-version file system are the disk-space and performance overheads. Since data is stored in N file systems, there is an N -fold increase (approximately) in disk space used. Since each file operation is performed on all file systems (except for file reads), the likely disk traffic is N times that for a single file system. For those environments where the user is willing to trade-off some data reliability for disk space and performance, we develop a variant of single-instance storage [11, 17, 35]. Note that SubSIST is not manda-

tory; if the performance and space overheads of EnvyFS are acceptable, there is no reason to make use of SubSIST (indeed, the less code relied upon the better).

With SubSIST, the disk operations of the multiple children pass through SubSIST, which is implemented as a block-level layer. As is common in single-instance stores, SubSIST computes a content hash (MD5) for all disk blocks being written and uses the content hash to detect duplicate data.

Using an SIS greatly reduces disk usage underneath an N-version file system. At the same time, despite coalescing data blocks, an SIS retains much of the benefit of EnvyFS for two reasons. First, the reliability of file-system *metadata* is not affected by the use of an SIS. Since metadata forms the access path to multiple units of data, its reliability may be considered more important than that of data blocks. Because the format of file-system metadata is different across different file systems, metadata blocks of different file systems have different hash values and are stored separately; thus, the SIS layer can distinguish between data and metadata blocks *without* any knowledge of file-system data structures. Second, since file systems maintain different in-memory copies of data, file-system bugs that corrupt data blocks in-memory cause the data in different file systems to have different content hashes; therefore, individual file systems are still protected against each other’s in-memory file-data corruptions.

4.1 Requirements and Implications

The design of SubSIST for an N-version file system should satisfy slightly different requirements than a conventional SIS. We discuss four important observations and their impact on the design of SubSIST.

First, child file systems often replicate important metadata blocks so that they can recover from failures. For example, JFS replicates its superblock and uses the replica to recover from a latent sector error to the primary. Thus, SubSIST does not coalesce disk blocks with the same content if they belong to the same file system.

Second, an SIS coalesces common data written at approximately the same time by different file systems. Therefore, in SubSIST, the content hash information for each disk block is not stored persistently; the content hashes are maintained in memory and deleted after some time has elapsed (or after N file systems have written the same content). This ephemeral nature of content hashes also reduces the probability of data loss or corruption due to hash collisions [10, 23].

Third, in an N-version file system, reads of the same data blocks occur at nearly the same time. Thus, SubSIST services reads from different file systems by maintaining a small read cache. This read cache holds only those disk blocks whose reference count (number of file

systems that use the block) is more than one. It also tracks the number of file systems that have read a block and removes a block from cache as soon as this number reaches the reference count for the block.

Finally, the child file systems using SubSIST are unmodified and therefore have no knowledge of content addressing; therefore, SubSIST virtualizes the disk address space; it exports a virtual disk to the file system, and maintains a mapping from each file system’s virtual disk address to the corresponding physical disk address, along with a reference count for each physical disk block. SubSIST uses file-system virtual addresses as well as previously mapped physical addresses as hints when assigning physical disk blocks to maintain as much sequentiality and spatial locality as possible. When these hints do not provide a free disk block, SubSIST selects the closest free block to the previously mapped physical block.

4.2 Implementation

SubSIST has numerous important data structures, including: (i) a table of virtual-to-physical mappings, (ii) allocation information for each physical disk block in the form of reference count maps, (iii) a content-hash cache of recent writes and the identities of the file systems that performed the write, and (iv) a small read cache.

We have built SubSIST as a pseudo-device driver in Linux. It exports virtual disks that are used by the file systems. Our current implementation does not store virtual-to-physical mappings and reference-count maps persistently; in the future, we plan to explore reliably writing this information to disk.

5 Reliability Evaluation

We evaluate the reliability improvements of a 3-version EnvyFS (EnvyFS_3) that uses ext3, JFS, and ReiserFS (v3) as children. All our experiments use the versions of these file systems that are available as part of the Linux 2.6.12 kernel.

We evaluate the reliability of EnvyFS_3 in two ways: First, we examine whether it recovers from scenarios where file-system content is different in one of the three children. Second, we examine whether it can recover from corruption to on-disk data structures of one child.

5.1 Differing File System Content

The first set of experiments is intended to mimic the scenario where one of the file systems has an incorrect disk image. Such a scenario might occur either when (i) a system crash occurs and one of the children has written more or less to disk than the others, (ii) a bug causes one of the file systems to corrupt file data, say by performing a misdirected write of data belonging to one file to another file, or (iii) soft memory errors cause corruption.

Difference in content	Num Tests	Correct success	Correct error code
None	28	17 / 17	11 / 11
Dir contents differ in one	13	6 / 6	7 / 7
Dir present in only two	13	6 / 6	7 / 7
Dir present in only one	9	4 / 4	5 / 5
File contents differ in one	15	11 / 11	4 / 4
File metadata differ in one	45	33 / 33	12 / 12
File present in only two	15	11 / 11	4 / 4
File present in only one	9	3 / 3	6 / 6
Total	147	91 / 91	56 / 56

Table 1: **File-system Content Experiments.** This table presents the results of issuing file operations to EnvyFS_3 objects that differ in data or metadata content across the different children. The first column describes the difference in file-system content. The second column presents the total number of experiments performed for this content difference; this is the number of applicable file operations for the file or directory object. For metadata differences, 15 operations each are performed for differences in mode, nlink, and size fields of the inode. The third column is the fraction of operations that return correct data and/or successfully complete. The fourth column is the fraction of operations that correctly return an error code (and it is the expected error code) (e.g., ENOENT when an unlink operation is performed for a non-existent file). We see that EnvyFS_3 successfully uses the majority result in all 147 experiments.

We first experiment by creating different file-system images as the children and executing a set of file operations on EnvyFS_3 that uses the children. We have explored various file-system content differences, including extra or missing files or directories, and differences in file or directory content. The different file operations performed include all possible file operations for the object (irrespective of whether the operation causes the different content to be read). Our file operations include those that are expected to succeed as well as those that are expected to fail with a specific error code.

Table 1 shows that EnvyFS_3 correctly detects all differences and always returns the majority result to the user (whether the expected data or error code). EnvyFS_3 can also be successfully mounted and unmounted in all cases. We find that the results are the same irrespective of which child (ext3, JFS, ReiserFS) has incorrect contents.

We then explore whether EnvyFS_3 continues to detect and recover from differences caused by in-memory corruption when SubSIST is added. We experiment by modifying data (or metadata) as it being written to a child file system and then causing the data (or metadata) to be read back. Table 2 presents the results of the experiments. We find that EnvyFS_3 used along with SubSIST returns the correct results in all scenarios. Also, in most sce-

Corruption Type	Num Tests	Correct success	Fix
File contents differ in one	3	3 / 3	3 / 3
Dir contents differ in one	3	3 / 3	0 / 3
Inode contents differ in one	15	15 / 15	9 / 15
Total	21	21 / 21	12 / 21

Table 2: **File-system Corruption Experiments.** This table presents the results of corrupting one of the file objects in EnvyFS₃ that results in different data or metadata content across the different children with SubSIST underneath it. The first column describes the type of corruption. The second column presents the total number of experiments performed; The third column is the fraction of operations that return correct data and/or successfully complete (which also include identification of mismatch in file system contents). The fourth column is the fraction of operations that EnvyFS was able to repair after detecting corruption.

narios when file contents or inode contents are different, EnvyFS₃ successfully repairs the corrupt child during file-system operation (Section 3.5.1 describes scenarios in which EnvyFS repairs a child during file-system operation). The use of SubSIST does not affect protection against in-memory corruption; a data block corrupted in memory will cause SubSIST to generate a different content hash for the bad block when it is written out, thereby avoiding the usual coalescing step.

5.2 Disk Corruption

The second set of experiments analyzes whether EnvyFS₃ recovers when a child’s on-disk data structures are corrupt. Such corruption may be due to a bug in the file system or the rest of the storage stack. We inject corruption into JFS and ext3 data structures by interposing a pseudo-device driver that has knowledge of the data structures of each file system. This driver zeroes the entire buffer being filled in response to a disk request by the file system, but does not return an error code (i.e., the corruption is silent). All results, except that for data blocks, are applicable to using EnvyFS₃ with SubSIST.

5.2.1 Corruption in JFS

Figures 2a and 2b compare the user-visible results of injecting corruptions into JFS data structures when JFS is used stand-alone and when EnvyFS₃ is used (that is composed of JFS, ext3, and ReiserFS).

Each row in the figures corresponds to the JFS data structure for which the fault is injected. Each column in the figures corresponds to different file operations. The different symbols represent the user-visible results of the fault; examples of user-visible results include data loss, and a non-mountable file system. For example, in Figure 2a, when an inode block is corrupted during path traversal (column 1), the symbol indicates that (i) the

operation fails and (ii) the file system is remounted in read-only mode. In addition to the symbols for each column, the symbol next to the data-structure name for all the rows indicates whether or not the loss of the disk block causes irreparable data or metadata loss.

As shown in Figure 2a, JFS is rarely able to recover from corruptions: JFS can continue normal operation when the read to the block-allocation bitmap fails during truncate and unlink. Often, the operation fails and JFS remounts the file system in read-only mode. The corruption of some data structures also results in a file system that cannot be mounted. In one interesting case, JFS detects the corruption to an internal (indirect) block of a file and remounts the file system in read-only mode, but still returns corrupt data to the user. Data loss is indicated for many of the JFS rows.

In comparison to stand-alone JFS, EnvyFS₃ recovers from all but one of the corruptions (Figure 2b). EnvyFS₃ detects errors reported by JFS and also detects corrupt data returned by JFS when the internal block or data block is corrupted during file read. In all these cases, EnvyFS₃ uses the two other file systems to continue normal operation. Therefore, no data loss occurs when any of the data structures is corrupted.

In one interesting fault-injection experiment, a system crash occurs both when using JFS stand-alone and when using it in EnvyFS₃. In this experiment, the first aggregate inode block (AGGR-INODE-1) is corrupted, and the actions of JFS lead to a kernel panic during paging. Since this call to panic is not in JFS code, it cannot be replaced as described in Section 3.4. Therefore, the kernel panic occurs both when using JFS stand-alone and when using EnvyFS₃. Thus, we find a case where EnvyFS₃ is not completely resilient to underlying child failure; faults that lead to subsequent panics in the main kernel cannot be handled with N-version techniques.

5.2.2 Corruption in Ext3

Figures 2c and 2d show the results of injecting corruption into ext3 data structures. As in the case of JFS, the figures compare ext3 against EnvyFS₃.

Overall, we find that ext3 does not handle corruption well. Figure 2c shows that no corruption error leads to normal operation without data loss for ext3. In most cases, there is unrecoverable data loss and either the operation fails (ext3 reports an error) or the file system is remounted in read-only mode or both. In some cases, the file system cannot even be mounted. In other cases, ext3 fails to detect corruption (e.g., IMAP, INDIRECT), thereby either causing data loss (IMAP) or returning corrupt data to the user (INDIRECT). Finally, in one scenario (corrupt INODE during unlink), the failure to handle corruption leads to a system crash upon unmount.

In comparison, Figure 2d shows that EnvyFS₃ contin-

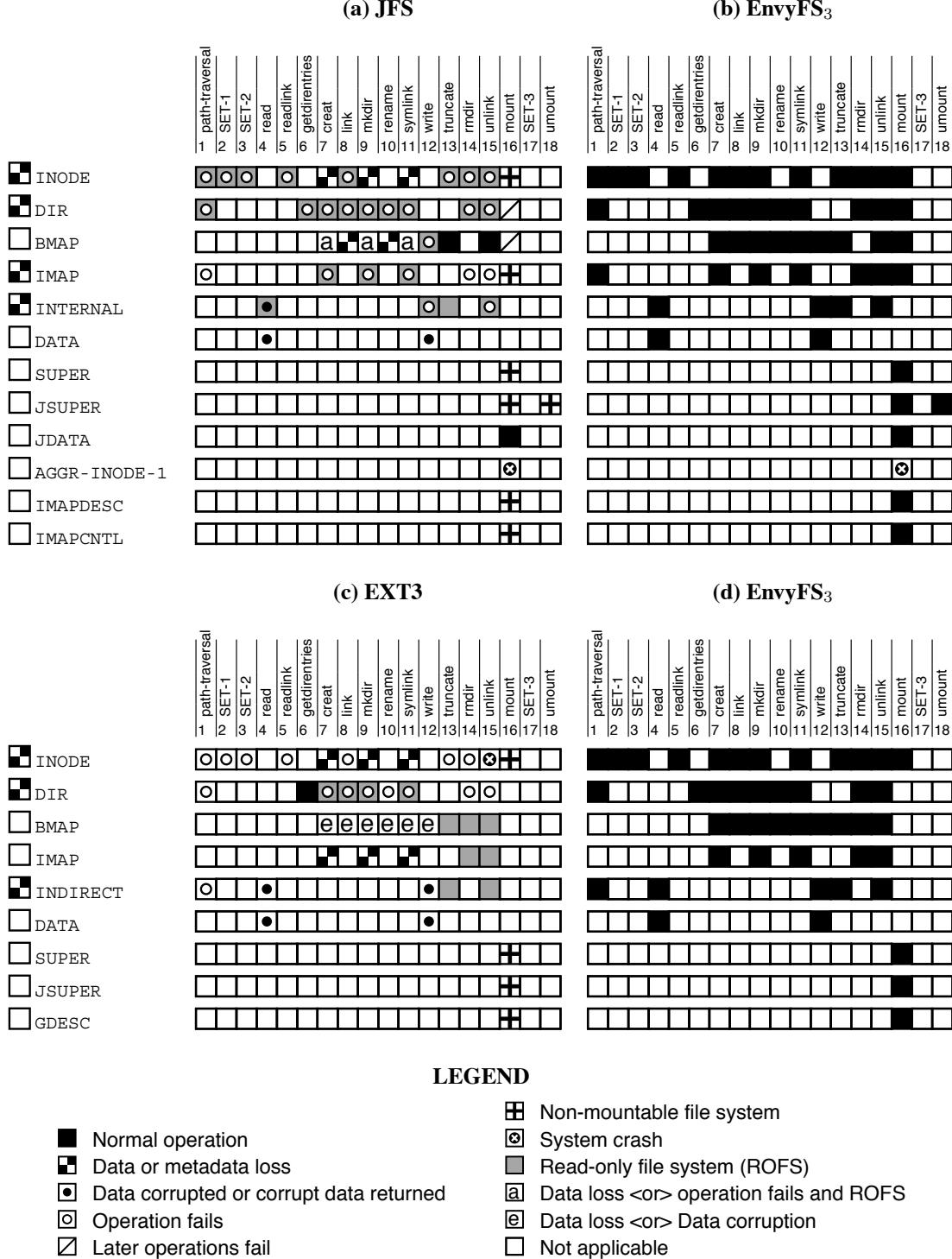


Figure 2: Disk corruption experiments. The figures show the results of injecting corruption for JFS and ext3 on-disk data structures. JFS is used stand-alone in (a) and is one of the children in EnvyFS₃ in (b). ext3 is used stand-alone in (c) and is one of the children in EnvyFS₃ in (d). Each row in the figures corresponds to the data structure for which the fault is injected; each column corresponds to a file operation; each symbol represents the user-visible result of the fault injection. Note that (i) the column SET-1 denotes file operations access, chdir, chroot, stat, statfs, lstat, and open; SET-2 denotes chmod, chown, and utimes; SET-3 denotes fsync and sync, (ii) some symbols are a combination of two symbols, one of which is the light-gray square for “read-only file system.”

ues normal operation in every single experiment, including in the system-crash case. EnvyFS₃ again shows great resilience to faults in a single child file system.

We also found EnvyFS₃ to be surprisingly helpful in isolating a non-trivial bug in ext3. As reported above, when an ext3 inode block is corrupted before an unlink, the system crashes when the file system is later unmounted. The system crash does not occur in EnvyFS₃; one might suspect that EnvyFS₃ is robust because ext3 was modified to call `nvfs_child_panic`. However, this is not the case; instead, EnvyFS₃ completely avoids the code paths that cause the panic; in particular, EnvyFS₃ detects that the inode returned by ext3 in response to a lookup (that is performed by VFS prior to the actual unlink) is faulty (i.e., semantically differs from the inodes returned by the other file systems). Therefore, it does not issue the subsequent unlink operation to ext3, hence avoiding actions that cause the panic. Interestingly, the bug that causes the crash is actually in the lookup operation, the first point where EnvyFS₃ detects a problem. Note that in the absence of an N-version file system, one would find that the system crashed on an unmount, but will not have information linking the crash to the unlink system call or the bug in `ext3_lookup`. Checking the ext3 source code, we found that this bug in Linux 2.6.12 was subsequently fixed in 2.6.23. This experience highlights the potential for using N-versioning to localize bugs in file systems.

5.3 Discussion

Our experiments show that EnvyFS₃ can recover from various kinds of corruptions in a child file system. Since this improvement in reliability is achieved through additional layers of code, any bugs in these layers could offset the reliability improvements. Therefore, an important goal in our design is to keep EnvyFS simple. We now compare the amount of code used to construct EnvyFS and SubSIST against other file systems in order to estimate the complexity (and therefore, the likelihood of bugs) in such a system.

The EnvyFS layer is about 3,900 lines of code, while SubSIST is about 2,500 lines of code. In comparison, ext3 contains 10,423 lines, JFS has 15,520 lines, ReiserFS has 18,537 lines, and XFS, a complex file system, has 44,153 lines.

6 Time and Space Overheads

Although reliable file-system operation is our major goal, we are also concerned with the overheads innate to an N-version approach. In this section, we quantify the performance costs of EnvyFS and the reduction in disk-space overheads due to SubSIST.

	ext3	JFS	Reiser	Envy₃	+SIS
Cached read	2.1	2.1	2.2	5.7	5.7
Cached write	3.7	2.5	2.2	8.8	8.8
Seq. read-4K	17.8	17.7	18.2	424.1	33.7
Seq. read-1M	17.8	17.7	18.2	75.4	33.7
Seq. write	26.0	18.7	24.4	74.9	29.7
Rand. read	163.6	163.5	165.1	434.2	164.2
Rand. write	20.4	18.9	20.4	61.4	7.0
OpenSSH	25.3	25.7	25.6	26.4	26.0
Postmark-10K	14.7	39.0	9.6	128.8	26.4
Postmark-100K	29.0	107.2	33.6	851.4	430.0
Postmark-100K*	128.3	242.5	78.3	405.5	271.1

Table 3: Performance. This table compares the execution time (in seconds) for various benchmarks for EnvyFS₃ (without and with SubSIST) against the child file systems, ext3, JFS, and ReiserFS. All our experiments use Linux 2.6.12 installed on a machine with an AMD Opteron 2.2 GHz processor, 2 GB RAM, Hitachi Deskstar 7200-rpm SATA disks, and 4-GB disk partitions for each file system. Cached reads and writes involve 1 million reads/writes to 1 file data block. Sequential read-4K/writes are 4 KB at a time to a 1-GB file. Sequential read-1M is 1MB at a time to a 1-GB file. Random reads/writes are 4 KB at a time to 100 MB of a 1-GB file. OpenSSH is a copy, untar, and make of OpenSSH-4.5. Postmark was configured to create 2500 files of sizes between 4KB and 40KB. We ran it with 10K and 100K transactions. All workloads except ones named “Cached” use a cold file-system cache.

We now quantify the performance overheads of EnvyFS₃ both with and without SubSIST, in contrast to each of the child file systems (ext3, JFS, and ReiserFS) running alone. Table 3 presents the results.

We now highlight the interesting points from the table:

- When reads hit in the cache (*cached reads*), EnvyFS₃ pays a little more than twice the cost (as it accesses data from only two children and performs a checksum comparison to find a majority).
- EnvyFS₃ performance under *cached writes* is roughly the sum across the children; such writes go to all three child file systems, and thus are replicated in the buffer cache three times. This aspect of EnvyFS₃ is bad for performance (and increases cache pressure), but at the same time increases fault resilience; a corruption to one copy of the data while in memory will not corrupt the other two copies.
- SubSIST does not help with either cached workload as it only interposes on disk traffic.
- EnvyFS₃ has terrible performance under *sequential disk reads*, as it induces seeks (and loses disk track prefetches) between two separate sequential streams especially with small block sizes; much of this cost could be alleviated with additional prefetching or with larger block sizes. Increasing

- the read size from 4KB to 1MB significantly improves the performance of EnvyFS₃.
- *Sequential writes* perform much better on EnvyFS₃ compared to sequential reads, due to batching of operations (and hence fewer seeks).
 - In many cases where EnvyFS₃ performance suffers (*sequential reads and writes, random reads*), SubSIST greatly improves performance through coalescing of I/O. Indeed, in one case (*random writes*), SubSIST improves performance of EnvyFS₃ as compared to any other single file system, as for this specific case its layout policy transforms random writes into a more sequential pattern to disk (see Section 4.1). These performance improvements likely represent the best case since the numbers do not show the costs that would be incurred in a SubSIST implementation that maintains data structures persistently.
 - Application performance, as measured on the *OpenSSH* benchmark, is quite acceptable, even without SubSIST.
 - In the case of *Postmark* benchmark, both workload size and dirty page writeout intervals affect the performance of EnvyFS₃. For smaller workloads (i.e., Postmark-10K), performance of EnvyFS₃ with SubSIST is comparable with other file systems. But with increase in workload size (Postmark-100K), performance of EnvyFS₃ worsens as it is forced to write back more data due to increase in cache pressure along with shorter dirty page writeout intervals. If we provide EnvyFS₃ with thrice the amount of memory and change the writeback intervals accordingly, we see that EnvyFS₃ performance (with SubSIST) is comparable to the slowest of the three children (JFS).

We also tracked the storage requirement across these benchmarks. For those workloads that generated writes to disk, we found that SubSIST reduced the storage requirement of EnvyFS by roughly a factor of three.

7 Related Work

Over the years, N-version programming has been used in various real systems and research prototypes to reduce the impact of software bugs on system reliability. As noted by Avižienis [1], N-version computing has very old roots (going back to Babbage and others in the 1800s).

The concept was (re)introduced in computer systems by Avižienis and Chen in 1977 [2]. Since then, various other efforts, many from the same research group, have explored the process as well as the efficacy of N-version programming [3, 5, 4, 13, 27].

Avižienis and Kelly [4] study the results of using different specification languages; they use 3 different specification languages to develop 18 different versions of an airport scheduler program. They perform 100 demanding transactions with different sets of 3-version units and determined that while at least one version failed in 55.1% of the tests, a collective failure occurred only in 19.9% of the cases. This demonstrates that the N-version approach reduces the chances of failure. Avižienis et al. also determine the usefulness of developing the different software versions in different languages like Pascal, C etc. [5]. As in the earlier study, the different versions developed had faults, but only very few of these faults were common and the source of the common faults were traced to ambiguities in the initial specification.

N-version computing has been employed in many systems. For many years, such uses have primarily been in mission-critical or safety-critical systems [48, 52]. More recently, with the increasing cost of system failures and the rising impact of software bugs, many research efforts have focused on solutions that use N-version programming for improving system security and for handling failures [15, 26, 37, 47]. Joukov et al. [26] store data across different local file systems with different options for storing the data redundantly. However, unlike our approach, they do not protect against file-system bugs, and inherently rely on each individual file system to report any errors, so that data recovery may be initiated in RAID-like fashion. Rodrigues et al. [37] develop a framework to allow the use heterogeneous network file systems as replicas for Byzantine-fault tolerance. Vandiver et al. [47] explore the use of heterogeneous database systems for Byzantine-fault tolerance. They specifically address the issue of ordering of operations using *commit barriers*. In EnvyFS, this issue is made simpler due to two reasons: (i) in the absence of transactions, file systems are not expected to provide atomicity across multiple operations on the same file, and (ii) the VFS layer can easily identify conflicts through locking of file-system data structures.

8 Conclusion

“A three-ply cord is not easily severed.”

King Solomon [Ecclesiastes 4:12]

We have proposed EnvyFS, an approach that harnesses the N-version approach to tolerate file-system bugs. Central to our approach is building a reliable whole out of existing and potentially unreliable parts, thereby significantly reducing the cost of development. We have also proposed the use of a single-instance store to reduce the performance and disk-space overheads of an N-version approach. SubSIST, the single-instance store,

is designed to retain much of the reliability improvements obtained from EnvyFS. We have built and evaluated EnvyFS for Linux file systems and shown that it is significantly more reliable than file systems of which it is composed; with SubSIST, performance and capacity overheads are brought into the acceptable range. As a fringe benefit, we also show that the N-version approach can be used to locate bugs in file systems.

Modern file systems are becoming more complex by the day; mechanisms to achieve data-structure consistency [45], scalability and flexible allocation of disk blocks [9, 42], and the capability to snapshot the file system [25, 40] significantly increase the amount of code and complexity in a file system. Such complexity could lead to bugs in the file system that render any data protection further down the storage stack useless. N-versioning can help; by building reliability on top of existing pieces, EnvyFS takes an end-to-end approach and thus delivers reliability in spite of the unreliability of the underlying components.

Of course, our approach is not a panacea. Each file system may have features that N-versioning hides or makes difficult to realize. For example, some file systems are tailored for specific workloads (e.g., LFS[38]). In the future, it would be interesting if one could enable the N-version layer to be cognizant of such differences; for example, if one file system is optimized for write performance, all writes could initially be directed to it, and only later (in the background) would other file systems be updated. In such a manner, we could truly achieve the best of both worlds: reliability of the N-version approach but without the loss of characteristics that makes each file system unique.

Acknowledgments

We thank the anonymous reviewers and Sean Rhea (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the ADSL research group for their insightful comments.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from NetApp, Inc and Sun Microsystems.

[†]Author is currently an employee of NetApp, Inc.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

- [1] A. A. Avižienis. The Methodology of N-Version Programming. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 2. John Wiley & Sons Ltd., 1995.
- [2] A. A. Avižienis and L. Chen. On the Implementation of N-Version Programming for Software Fault Tolerance During Ex-

ecution. In *Proceedings of 1st Annual International Computer Software and Applications Conference (COMPSAC'77)*, Chicago, USA, 1977.

- [3] A. A. Avižienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges. The UCLA DEDIX system: A Distributed Testbed for Multiple-version Software. In *Digest of 15th International Symposium on Fault-Tolerant Computing (FTCS'85)*, pages 126–134, Ann Arbor, MI, June 1985.
- [4] A. A. Avižienis and J. P. J. Kelly. Fault Tolerance by Design Diversity: Concepts and Experiments. *IEEE Computer*, 17(8), August 1984.
- [5] A. A. Avižienis, M. R. Lyu, and W. Schütz. In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software. In *Digest of 18th International Symposium on Fault-Tolerant Computing (FTCS '88)*, Tokyo, Japan, June 1988.
- [6] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.
- [7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpacı-Dusseau, and R. H. Arpacı-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, California, February 2008.
- [8] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.
- [9] S. Best. JFS Overview. <http://jfs.sourceforge.net/project/pub/jfs.pdf>, 2000.
- [10] J. Black. Compare-by-hash: a reasoned analysis. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, pages 7–12, Boston, Massachusetts, June 2006.
- [11] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, Washington, August 2000.
- [12] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.
- [13] L. Chen and A. A. Avižienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Digest of 8th International Symposium on Fault-Tolerant Computing (FTCS'78)*, Toulouse, France, 1978.
- [14] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.
- [15] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davdison, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-Variant Systems - A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium (Sec '06)*, Vancouver, British Columbia, Aug. 2006.
- [16] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.
- [17] EMC. Centera Family. <http://www.emc.com/products/family/emc-centera-family.htm>, 2009.
- [18] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

- [19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [20] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.
- [21] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.
- [22] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994.
- [23] V. Henson. An Analysis of Compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03)*, Lihue, Hawaii, May 2003.
- [24] V. Henson. The Many Faces of fsck. <http://lwn.net/Articles/248180/>, September 2007.
- [25] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.
- [26] N. Joukov, A. Rai, and E. Zadok. Increasing Distributed Storage Survivability with a Stackable RAID-like File System. In *Proceedings of the 1st International Workshop on Cluster Security (Cluster-Sec'05)*, Cardiff, UK, 2005.
- [27] J. P. J. Kelly and A. A. Avižienis. A Specification-Oriented Multi-version Software Experiment. In *Digest of 13th International Symposium on Fault-Tolerant Computing (FTCS '83)*, Milano, Italy, June 1983.
- [28] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, California, February 2008.
- [29] Z. Li, Z. Chen, S. M. Srivivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 173–186, San Francisco, California, April 2004.
- [30] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.
- [31] D. Milošić, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing Relevance of Memory Hardware Errors: A Case for Recoverable Programming Models. In *9th ACM SIGOPS European Workshop 'Beyond the PC: New Challenges for the Operating System'*, Kolding, Denmark, September 2000.
- [32] B. Pawłowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS Version 4 Protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, May 2000.
- [33] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, pages 802–811, Yokohama, Japan, June 2005.
- [34] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.
- [35] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.
- [36] H. Reiser. ReiserFS. www.namesys.com, 2004.
- [37] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.
- [38] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [39] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2007.
- [40] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.
- [41] R. Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matrl/sample_0206tot_resiliency.html, February 2006.
- [42] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.
- [43] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.
- [44] M. M. Swift, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 1–16, San Francisco, California, December 2004.
- [45] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.
- [46] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.
- [47] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine Faults in Transaction Processing Systems using Commit Barrier Scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, Washington, October 2007.
- [48] U. Voges, editor. *Software Diversity in Computerized Control Systems*. Springer, Wien, New York, Dec. 1988.
- [49] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.
- [50] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically Generating Malicious Disks using Symbolic Execution. In *IEEE Security and Privacy (SP '06)*, Berkeley, California, May 2006.
- [51] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.
- [52] Y. C. Yeh. Triple-Triple Redundant 777 Primary Flight Computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, 1996.

Decentralized Deduplication in SAN Cluster File Systems

Austin T. Clements*

Irfan Ahmad

VMware, Inc.

Murali Vilayannur

Jinyuan Li

**MIT CSAIL*

Abstract

File systems hosting virtual machines typically contain many duplicated blocks of data resulting in wasted storage space and increased storage array cache footprint. *Deduplication* addresses these problems by storing a single instance of each unique data block and sharing it between all original sources of that data. While deduplication is well understood for file systems with a centralized component, we investigate it in a decentralized cluster file system, specifically in the context of VM storage.

We propose DEDE, a block-level deduplication system for live cluster file systems that does not require any central coordination, tolerates host failures, and takes advantage of the block layout policies of an existing cluster file system. In DEDE, hosts keep summaries of their own writes to the cluster file system in shared on-disk logs. Each host periodically and independently processes the summaries of its locked files, merges them with a shared index of blocks, and reclaims any duplicate blocks. DEDE manipulates metadata using general file system interfaces without knowledge of the file system implementation. We present the design, implementation, and evaluation of our techniques in the context of VMware ESX Server. Our results show an 80% reduction in space with minor performance overhead for realistic workloads.

1 Introduction

Deployments of consolidated storage using Storage Area Networks (SANs) are increasing, motivated by universal access to data from anywhere, ease of backup, flexibility in provisioning, and centralized administration. SAN arrays already form the backbone of modern data centers by providing consolidated data access for multiple hosts simultaneously. This trend is further fueled by the proliferation of virtualization technologies, which rely on shared storage to support features such as live migration of virtual machines (VMs) across hosts.

SANs provide multiple hosts with direct SCSI access to shared storage volumes. Regular file systems assume exclusive access to the disk and would quickly corrupt a shared disk. To tackle this, numerous shared disk cluster file systems have been developed, including VMware VMFS [21], RedHat GFS [15], and IBM GPFS [18], which use distributed locking to coordinate concurrent access between multiple hosts.

Cluster file systems play an important role in virtualized data centers, where multiple physical hosts each run potentially hundreds of virtual machines whose virtual disks are stored as regular files in the shared file system. SANs provide hosts access to shared storage for VM disks with near native SCSI performance while also enabling advanced features like live migration, load balancing, and failover of VMs across hosts.

These shared file systems represent an excellent opportunity for detecting and coalescing duplicate data. Since they store data from multiple hosts, not only do they contain more data, but data redundancy is also more likely. Shared storage for VMs is a ripe application for deduplication because common system and application files are repeated across VM disk images and hosts can automatically and transparently share data between and within VMs. This is especially true of virtual desktop infrastructures (VDI) [24], where desktop machines are virtualized, consolidated into data centers, and accessed via thin clients. Our experiments show that a real enterprise VDI deployment can expend as much as 80% of its overall storage footprint on duplicate data from VM disk images. Given the desire to lower costs, such waste provides motivation to reduce the storage needs of virtual machines both in general and for VDI in particular.

Existing deduplication techniques [1, 3–5, 8, 14, 16, 17, 26] rely on centralized file systems, require cross-host communication for critical file system operations, perform deduplication in-band, or use content-addressable storage. All of these approaches have limitations in our domain. Centralized techniques would be difficult to ex-

tend to a setting with no centralized component other than the disk itself. Existing decentralized techniques require cross-host communication for most operations, often including reads. Performing deduplication in-band with writes to a live file system can degrade overall system bandwidth and increase IO latency. Finally, content-addressable storage, where data is addressed by its content hash, also suffers from performance issues related to expensive metadata lookups as well as loss of spatial locality [10].

Our work addresses deduplication in the decentralized setting of VMware’s VMFS cluster file system. Unlike existing solutions, DEDE coordinates a cluster of hosts to cooperatively perform block-level deduplication of the live, shared file system. It takes advantage of the shared disk as the only centralized point in the system and does not require cross-host communication for regular file system operations, retaining the direct-access advantage of SAN file systems. As a result, the only failure that can stop deduplication is a failure of the SAN itself, without which there is no file system to deduplicate. Because DEDE is an online system for primary storage, all deduplication is best-effort and performed as a background process, out-of-band from writes, in order to minimize impact on system performance. Finally, unlike other systems, DEDE builds block-level deduplication atop an existing file system and takes advantage of regular file system abstractions, layout policy, and block addressing. As a result, deduplication introduces no additional metadata IO when reading blocks and permits in-place writes to blocks that have no duplicates.

This paper presents the design of DEDE. We have implemented a functional prototype of DEDE for VMware ESX Server [23] atop VMware VMFS. Using a variety of synthetic and realistic workloads, including data from an active corporate VDI installation, we demonstrate that DEDE can reduce VM storage requirements by upwards of 80% at a modest performance overhead.

Section 2 provides an overview of the architecture of our system and our goals. Section 3 details the system’s design and implementation. We provide a quantitative evaluation of our system in Section 4, followed by a discussion of related work in Section 5. Finally, we conclude in Section 6.

2 System Overview

DEDE operates in a cluster setting, as shown in Figure 1, in which multiple hosts are directly connected to a single, shared SCSI volume and use a file system designed to permit symmetric and cooperative access to the data stored on the shared disk. DEDE itself runs on each host as a layer on top of the file system, taking advantage of file system block layout policies and native support for

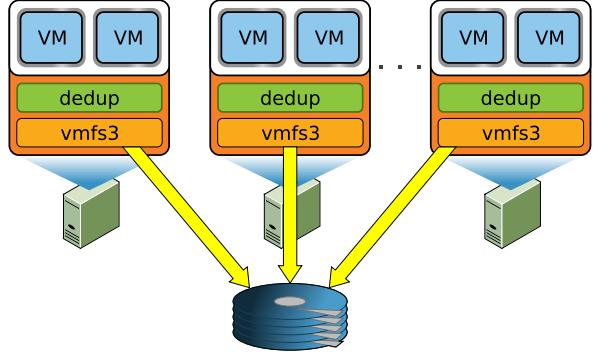


Figure 1: Cluster configuration in which multiple hosts concurrently access the same storage volume. Each host runs the VMFS file system driver (`vmfs3`), the deduplication driver (`dedup`), and other processes such as VMs.

copy-on-write (COW) blocks. In this section, we provide a brief overview of our approach to deduplication and the file system support it depends on.

DEDE uses content hashes to identify potential duplicates, the same basic premise shared by all deduplication systems. An index stored on the shared file system and designed for concurrent access permits efficient duplicate detection by tracking all known blocks in the file system by their content hashes.

In order to minimize impact on critical file system operations such as reading and writing to files, DEDE updates this index *out of band*, buffering updates and applying them in large, periodic batches. As part of this process, DEDE detects and eliminates duplicates introduced since the last index update. This can be done as an infrequent, low priority background task or even scheduled during times of low activity. Unlike approaches to deduplication such as content-addressable storage that integrate content indexes directly into the file system storage management, DEDE’s index serves solely to identify duplicate blocks and plays no role in general file system operations.

DEDE divides this index update process between hosts. Each host monitors its own changes to files in the cluster file system and stores summaries of recent modifications in on-disk *write logs*. These logs include content hashes computed in-band, as blocks are written to disk. Each host periodically consumes the write logs of files it has (or can gain) exclusive access to and updates the shared index to reflect these recorded modifications. In the process, it discovers and reclaims any block whose content is identical to the content of some previously indexed block. Having each host participate in the index update process allows the hosts to divide and distribute the burden of deduplication, while sharing the index allows hosts to detect duplicates even if they are introduced by separate hosts.

Out-of-band index updates mean DEDE must be resilient to stale index entries that do not reflect the latest content of recently updated blocks. Indeed, this is essentially unavoidable in a decentralized setting because of communication delays alone. While this means DEDE generally must verify block contents when updating the index, this resilience has an important implication: DEDE’s correctness does not depend on its ability to monitor every write to the file system. This has important performance benefits. First, updates to write logs do not have to be crash-consistent with updates to file contents, which both simplifies fault tolerance and allows hosts to buffer updates to write logs to minimize additional IO. Second, this allows users to trade off the CPU and memory overhead of write monitoring for peak file system performance on a per-file basis. For example, a user could simply disable deduplication for VMs that are performance-critical or unlikely to contain much duplicate data. Finally, this allows the write monitor to shed work if the system is overloaded.

Because DEDE operates on a live file system, it specifically optimizes for *unique* blocks (blocks with no known duplicates). Unlike *shared* blocks, these blocks remain mutable after deduplication. The mutability of unique blocks combined with DEDE’s resilience to stale index information means these blocks can be updated in place without the need to allocate space for a copy or to synchronously update the index. As a result, deduplication has no impact on the performance of writing to unique blocks, a highly desirable property because these are precisely the blocks that do not benefit from deduplication.

Similar to some other deduplication work related to virtual disks [10, 13], DEDE uses fixed-size blocks. Unlike stream-oriented workloads such as backup, where variable-sized chunks typically achieve better deduplication [26], our input data is expected to be block-structured because guest file systems (*e.g.*, ext3, NTFS) typically divide the disk into fixed-size 4 KB or 8 KB blocks themselves. Consistent with this expectation, earlier work [12] and our own test results (see Section 4.1), we use a block size of 4 KB.

2.1 Required File System Abstractions

Most approaches to deduplication unify duplicate elimination and storage management, supplanting the file system entirely. DEDE, in contrast, runs as a layer on top of VMFS, an existing file system. This layer finds potentially identical blocks and identifies them to the file system, which is then responsible for merging these blocks into shared, copy-on-write blocks.

DEDE requires the file system to be block oriented and to support file-level locking. The file system block size must also align with the deduplication block size, a

requirement VMFS’s default 1 MB block size, unfortunately, does not satisfy. Our only non-trivial change to VMFS was to add support for typical file system block sizes (*i.e.*, 4 KB), as detailed later in Section 2.2.

Finally, DEDE requires block-level copy-on-write support, a well understood, but nevertheless uncommon feature supported by VMFS. Specifically, it requires an unusual *compare-and-share* operation, which replaces two blocks with one copy-on-write block after verifying that the blocks are, in fact, identical (using either bit-wise comparison or a content hash witness). Despite the specificity of this operation, it fits naturally into the structure of block-level copy-on-write and was easy to add to the VMFS interface. DEDE manipulates file system blocks solely through this interface and has no knowledge of the underlying file system representation.

There are two noteworthy capabilities that DEDE does *not* require of the file system. First, hosts running DEDE never modify the metadata of files they do not have exclusive locks on, as doing so would require cross-host synchronization and would complicate per-host metadata caching. As a result, a host that discovers a duplicate block between two files cannot simply modify both files to point to the same block if one of the files is locked by another host. Instead, when DEDE detects a duplicate between files locked by different hosts, it uses a third file containing a *merge request* as an intermediary. One host creates a merge request containing a COW reference to the deduplicated block, then passes ownership of the merge request file’s lock to the other host, which in turn replaces the block in its file with a reference to the block carried by the merge request.

Second, DEDE does *not* require the file system to expose a representation of block addresses. Much like any regular application, it only refers to blocks indirectly, by their offset in some locked file, which the file system can resolve into a block address. This restricts the design of our index, since it cannot simply refer to indexed blocks directly. However, this limitation simplifies our overall design, since requiring the file system to expose block addresses outside the file system’s own data structures would interfere with its ability to free and migrate blocks and could result in dangling pointers. Worse, any operations introduced to manipulate blocks directly would conflict with file-level locking and host metadata caching.

In lieu of referring to blocks by block addresses, DEDE introduces a *virtual arena* file. This is a regular file in the file system, but it consists solely of COW references to shared blocks that are present in at least one other file. This file acts as an alternate view of all shared blocks in the system: DEDE identifies shared blocks simply by their offsets in the virtual arena file, which the file system can internally resolve to block addresses using regular address resolution.

Because DEDE builds on the underlying file system, it inherits the file system’s block placement policy and heuristics. If the underlying file system keeps file blocks sequential, blocks will generally remain sequential after deduplication. Shared blocks are likely to be sequential with respect to other blocks in at least one file, and common sequences of shared blocks are likely to remain sequential with respect to each other. Furthermore, the placement and thus sequentiality of unique blocks is completely unaffected by the deduplication process; as a result, deduplication does not affect IO performance to individual unique blocks because they do not require copying, and it maintains sequential IO performance across spans of unique blocks.

2.2 VMFS

Many of the design decisions in DEDE were influenced by the design of its substrate file system, VMFS. VMFS is a coordinator-less cluster file system [21] designed to allow hosts to cooperatively maintain a file system stored on a shared disk. In this section, we provide a quick overview of how VMFS addresses and manages concurrent access to its resources in order to provide better context for the design of DEDE.

VMFS organizes the shared disk into four different resource pools: inodes, pointer blocks, file blocks, and sub-blocks. Inodes and pointer blocks play much the same role as in traditional UNIX file systems, storing per-file metadata and pointers to the blocks containing actual file content. File blocks and sub-blocks both store file content, but are different sizes, as discussed below. The divisions between these pools are currently fixed at format time and can only be expanded by adding more storage, though this is not a fundamental limitation. In each pool, resources are grouped into *clusters*. The header for each cluster maintains metadata about all of its contained resources; most importantly, this includes a reference count for each individual resource and tracks which resources are free and which are allocated.

In order to support concurrent access by multiple hosts to file and resource data, VMFS uses a distributed lock manager. Unlike most cluster file systems, which use an IP network for synchronization, VMFS synchronizes all file system accesses entirely through the shared disk itself using on-disk locks. VMFS ensures atomic access to on-disk lock structures themselves using SCSI-2-based LUN reservations to guard read-modify-write critical sections. In addition to taking advantage of the reliability of storage area networks, using the same means to access both file system state and synchronization state prevents “split brain” problems typical of IP-based lock managers in which multiple hosts can access the file system state but cannot communicate locking decisions with each other.

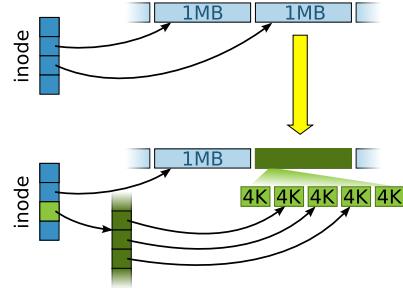


Figure 2: Mixed block sizes allow any 1 MB file block to be divided into 256 separate 4 KB sub-blocks.

VMFS protects file data from concurrent access by associating a coarse-grain lock with each file that covers all of a file’s metadata (its inode and pointer blocks) as well as all of the file blocks and sub-blocks comprising the file’s content. Files in VMFS tend to be locked for long durations (*e.g.*, a VM’s disk files are locked as long as the VM is powered on). DEDE respects file system locking by partitioning the deduplication process according to which hosts hold which file locks.

VMFS protects resource metadata using per-cluster locks. Thus, allocation and deallocation of resources must lock all clusters containing any of the resources involved. The number of resources packed per cluster reflects a trade-off between locking overhead and cross-host cluster lock contention. Higher cluster density allows hosts to manipulate more resources with fewer locks, but at the cost of increased lock contention. Since DEDE stresses the sub-block resource pool more than typical VMFS usage, we increase the sub-block cluster density from 16 to 128 resources per cluster, but otherwise use the default VMFS densities.

VMFS maintains two separate resource types for storing file content: file blocks and sub-blocks. File sizes in VMFS typically fit a bimodal distribution. Virtual machine disks and swap files are usually several gigabytes, while configuration and log files tend to be a few kilobytes. Because of this, VMFS uses 1 MB file blocks to reduce metadata overhead and external fragmentation for large files, while for small files, VMFS uses smaller sub-blocks to minimize internal fragmentation. DEDE must be able to address individual 4 KB blocks in order to COW share them, so we configure VMFS with 4 KB sub-blocks. Furthermore, rather than simply eschewing the efficiency of 1 MB blocks and storing all file content in 4 KB blocks, we extend VMFS to support *mixed block sizes*, depicted in Figure 2, so that DEDE can address individual 4 KB blocks of a file when it needs to share a duplicate block, but when possible still store unique regions of files in efficient 1 MB blocks. This change introduces an optional additional pointer block level and

allows any file block-sized region to be broken into 256 separate 4 KB blocks, which, in turn, add up to the original file block. This can be done dynamically to any 1 MB block based on deduplication decisions, and leaves address resolution for other data intact and efficient.

Beyond these unusual block sizes, VMFS supports a number of other uncommon features. Most important to DEDE is support for block-level copy-on-write (COW). Each file or sub-block resource can be referenced from multiple pointer blocks, allowing the same data to be shared between multiple places in multiple files. Each reference to a shared resource is marked with a COW bit, indicating that any attempts to write to the resource must make a private copy in a freshly allocated resource and write to that copy instead. Notably, this COW bit is associated with each *pointer* to the resource, not with the resource itself. Otherwise, every write operation would need to take a cluster lock to check the COW bit of the destination block, even if the block was not COW. However, as a result, sharing a block between two files requires file locks on *both* files, even though only one of the references will change. Thus, DEDE must use merge requests for all cross-host merging operations.

VMFS forms the underlying substrate of DEDE and handles critical correctness requirements such as specializing COW blocks and verifying potential duplicates, allowing DEDE to focus on duplicate detection. Virtual arenas and merge requests allow DEDE to achieve complex, decentralized manipulations of the file system structure without knowledge of the file system representation, instead using only a few general-purpose interfaces.

3 Design and Implementation

In this section, we provide details of the design and implementation of DEDE’s best-effort write monitoring subsystem and the out-of-band indexing and duplicate elimination process.

3.1 Write Monitoring

Each host runs a *write monitor*, as shown in Figure 3, which consists of a lightweight kernel module (*dedup*) that monitors all writes by that host to files in the file system and a userspace daemon (*dedupd*) that records this information to logs stored in the shared file system. The write monitor is the only part of the system that lies in the IO critical path of the file system, so the write monitor itself must incur as little additional disk IO and CPU overhead as possible.

The kernel module provides the userspace daemon with a modification stream indicating, for each write done by the host: the file modified, the offset of the write, and

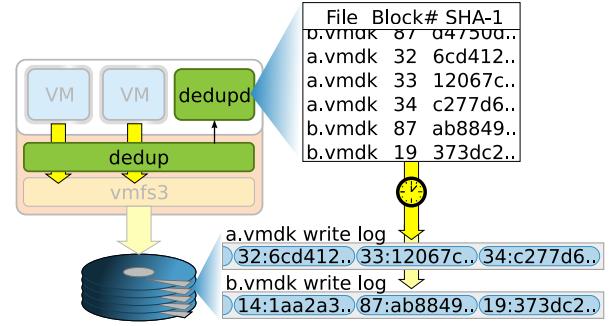


Figure 3: Only a lightweight kernel module lies in the IO critical path, opportunistically calculating hashes of blocks while they are still in memory. A userspace daemon (*dedupd*) flushes write logs to disk periodically. Duplicate detection and elimination occur out of band.

the SHA-1 hashes of all modified blocks. While the in-band CPU overhead of the monitor could have been virtually eliminated by computing these hashes lazily (*e.g.*, at indexing time), this would have required reading the modified blocks back from disk, resulting in a large amount of additional random IO. We opted instead to eliminate the extra IO by computing these hashes while the blocks were in memory, though the trade-off between run-time CPU overhead and deduplication-time IO overhead could be set dynamically by user-defined policy.

The userspace daemon divides the modification stream by file, aggregates repeated writes to the same block, and buffers this information in memory, periodically flushing it to individual write log files associated with each regular file. These write logs are stored on the shared file system itself, so even if a host fails or transfers ownership of a file’s lock, any other host in the system is capable of reading logs produced by that host and merging information about modified blocks into the index.

The daemon can safely buffer the modification stream in memory because the index update process is designed to deal with stale information. Without this, write logs would have to be consistent with on-disk file state, and each logical write to the file system would result in at least two writes to the disk. Instead, buffering allows our system to absorb writes to over 150 MB of file blocks into a single infrequent 1 MB sequential write to a log file. This is the only additional IO introduced by the write monitor.

Similarly, we rely on the best-effort property of write monitoring to minimize IO in the case of partial block writes. If a write to the file system does not cover an entire block, the monitor simply ignores that write, rather than reading the remainder of the block from disk simply to compute its hash. In practice, this is rarely a problem when writes originate from a virtual machine, because

guest operating systems typically write whole guest file system blocks, which are generally at least 4 KB.¹

Write monitoring can be enabled or disabled per file. If the performance of some VM is too critical to incur the overhead of write monitoring or if the system administrator has a priori knowledge that a VM’s duplication ratio is small, such VMs can be opted out of deduplication.

3.2 The Index

The shared on-disk index tracks all known blocks in the file system by their content hashes. As discussed in Section 2, each host updates this index independently, incorporating information about recent block modifications from the write logs in large batches on a schedule set by user-defined policy (*e.g.*, only during off-peak hours). A match between a content hash in the index and that of a recently modified block indicates a potential duplicate that must be verified and replaced with a copy-on-write reference to the shared block.

The index acts as an efficient map from hashes to block locations. Because DEDE treats unique blocks (those with only a single reference) differently from shared blocks (those with multiple references), each index entry can likewise be in one of two states, denoted $\text{Unique}(H, f, o)$ and $\text{Shared}(H, a)$. An index entry identifies a unique block with hash H by the inumber f of its containing file and its offset o within that file. Because index updates are out-of-band and unique blocks are mutable, these entries are only *hints* about a block’s hash. Thus, because a mutable block’s contents may have changed since it was last indexed, its contents must be verified prior to deduplicating it with another block. Shared blocks, on the other hand, are marked COW and thus their content is guaranteed to be stable. The index identifies each shared block by its offset a in the index’s *virtual arena*, discussed in the next section.

3.2.1 Virtual Arena

When duplicate content is found, DEDE reclaims all but one of the duplicates and shares that block copy-on-write between files. Because hosts can make per-file, mutable copies of shared blocks at any time without updating the index, we cannot simply identify shared blocks by their locations in deduplicated files, like we could for unique blocks. The index needs a way to refer to these shared blocks that is stable despite shifting references from deduplicated files. As discussed earlier, DEDE cannot simply store raw block addresses in the index because exposing these from the file system presents numerous problems.

¹Unfortunately, owing to an ancient design flaw in IBM PC partition tables, guest writes are not necessarily *aligned* with DEDE blocks. Section 4.1 has a more detailed analysis of this.

Instead, we introduce a virtual arena file as an additional layer of indirection that provides stable identifiers for shared blocks without violating file system abstractions.

The virtual arena is a regular file, but unlike typical files, it doesn’t have any data blocks allocated specifically for it (hence, it is virtual). Rather, it serves as an alternate view of all shared blocks in the file system. In this way, it is very different from the arenas used in other deduplication systems such as Venti [16], which store actual data blocks addressed by content addresses.

In order to make a block shared, a host introduces an additional COW reference to that block from the virtual arena file, using the same interface that allows blocks to be shared between any two files. Apart from uncollected garbage blocks, the virtual arena consumes only the space of its inode and any necessary pointer blocks. Furthermore, this approach takes advantage of the file system’s block placement policies: adding a block to the virtual arena does *not* move it on disk, so it is likely to remain sequential with the original file.

The index can then refer to any shared block by its *offset* in the virtual arena file, which the file system can internally resolve to a block address, just as it would for any other file. The virtual arena file’s inode and pointer block structure exactly form the necessary map from the abstract, stable block identifiers required by the index to the block addresses required by the file system.

3.2.2 On-disk Index Representation

DEDE stores the index on disk as a packed list of entries, sorted by hash. Because DEDE always updates the index in large batches and since the hashes of updates exhibit no spatial locality, our update process simply scans the entire index file linearly in tandem with a sorted list of updates, merging the two lists to produce a new index file. Despite the simplicity of this approach, it outperforms common index structures optimized for individual random accesses (*e.g.*, hash tables and B-trees) even if the update batch size is small. Given an average index entry size of b bytes, a sequential IO rate of s bytes per second, and an average seek time of k seconds, the time required to apply U updates using random access is Uk , whereas the time to scan and rewrite an index of I entries sequentially is $2lb/s$. If the ratio of the batch size to the index size exceeds $U/I = 2b/sk$, sequentially rewriting the entire index is faster than applying each update individually. For example, given an entry size of 23 bytes and assuming a respectable SAN array capable of 150 MB/s and 8 ms seeks, the batch size only needs to exceed 0.004% of the index size. Furthermore, hosts defer index updates until the batch size exceeds some fixed fraction of the index size (at least 0.004%), so the amortized update cost remains constant regardless of index size.

In order to allow access to the index to scale with the number of hosts sharing the file system, while still relying on file locking to prevent conflicting index access, hosts *shard* the index into multiple files, each representing some subdivision of the hash space. Once the time a host takes to update a shard exceeds some threshold, the next host to update that shard will split the hash range covered by the shard in half and write out the two resulting sub-shards in separate files. This technique mirrors that of extensible hashing [6], but instead of bounding the size of hash buckets, we bound the time required to update them. Combined with file locking, this dynamically adjusts the concurrency of the index to match demand.

3.3 Indexing and Duplicate Elimination

As the index update process incorporates information about recently modified blocks recorded in the write logs, in addition to detecting hash matches that indicate potential duplicates, it also performs the actual COW sharing operations to eliminate these duplicates. The duplicate elimination process must be interleaved with the index scanning process because the results of block content verification can affect the resulting index entries.

In order to update the index, a host sorts the recent write records by hash and traverses this sorted list of write records in tandem with the sorted entries in the index. A matching hash between the two indicates a potential duplicate, which is handled differently depending on the state of the matching index entry. Figure 4 gives an overview of all possible transitions a matching index entry can undergo, given its current state.

When DEDE detects a potential duplicate, it depends on the file system's compare-and-share operation, described in Section 2.1, to atomically verify that the block's content has not changed and replace it with a COW reference to another block. Based on user-specified policy, this verification can either be done by reading the contents of the potential duplicate block and ensuring that it matches the expected hash (*i.e.*, compare-by-hash), or by reading the contents of *both* blocks and performing a bit-wise comparison (*i.e.*, compare-by-value). If the latter policy is in effect, hash collisions reduce DEDE's effectiveness, but do *not* affect its correctness. Furthermore, because hashes are used solely for finding potential duplicates, if SHA-1 is ever broken, DEDE has the unique capability of gracefully switching to a different hash function by simply rebuilding its index. The content verification step can be skipped altogether if a host can prove that a block has not changed; for example, if it has held the lock on the file containing the block for the entire duration since the write record was generated and no write records have been dropped. While this is a fairly specific condition, it is often met in DEDE's target

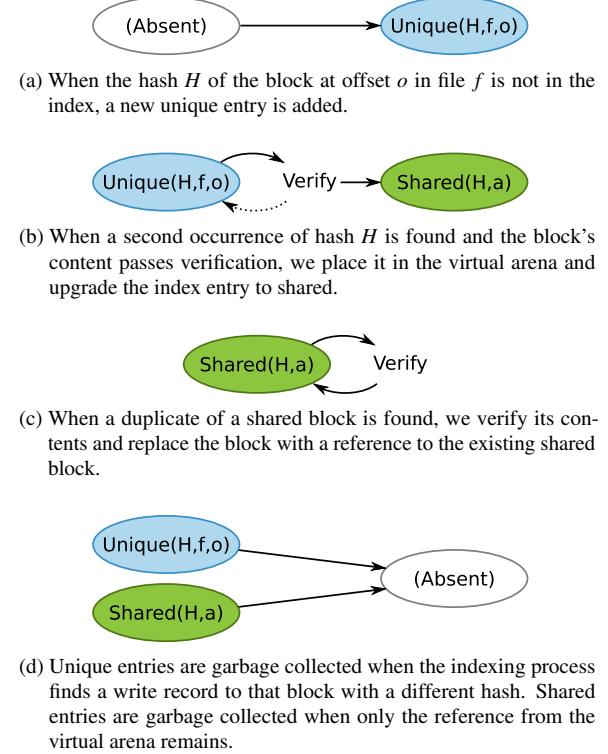


Figure 4: All possible updates to an index entry.

setting because locks on VM disks are usually held for very long durations.

3.3.1 Single Host Indexing

We begin with an explanation of the index update process assuming only a single host with exclusive access to the file system. In a single host design, the host can modify the metadata of any file. We lift this assumption in the next section, where we extend the process to support multiple hosts.

Any write record without a corresponding hash in the index indicates a new, unique block. Even though this write record may be stale, because index entries for unique blocks are only hints, it is safe to simply add the new unique block to the index without verifying the block's content, performing an *absent-to-unique* transition as shown in Figure 4(a). This single sequential, buffered write to the index is the only IO incurred when processing a new unique block.

When a write record's hash corresponds to an index entry for a unique block, then the host attempts to share both blocks (freeing one of them in the process) and upgrade the index entry to refer to the shared block. This *unique-to-shared* transition is shown in Figure 4(b). However, because the write record and index entry may both be stale, the host must verify the contents of both blocks before ac-

tually sharing them. Assuming this verification succeeds, the file system replaces both blocks with a shared block and the host inserts this block into the virtual arena and upgrades the index entry to refer to the new, shared block.

Finally, if a write record’s hash matches an index entry for a shared block, then the host attempts to eliminate this newly detected potential duplicate, performing a *shared-to-shared* transition as shown in Figure 4(c). Because the write record may be stale, it first verifies that the content of the potential duplicate has not changed. If this succeeds, then this block is freed and the reference to the block is replaced with a reference to the shared block found via the virtual arena.

3.3.2 Multi-Host Indexing

Extending the index update process to multiple hosts, we can no longer assume that a host will have unfettered access to every file. In particular, hosts can only verify blocks and modify block pointers in files they hold exclusive locks on. As a result, indexing *must* be distributed across hosts. At the same time, we must minimize communication between hosts, given the cost of communicating via the shared disk. Thus, sharing of blocks is done without any blocking communication between hosts, even if the blocks involved are in use by different hosts.

In the multi-host setting, the write logs are divided amongst the hosts according to which files each host has (or can gain) exclusive access to. While this is necessary because hosts can only process write records from files they hold exclusive locks on, it also serves to divide the deduplication workload between the hosts.

Absent-to-unique transitions and shared-to-shared transitions are the same in the multi-host setting as in the single host setting. Adding a new, unique block to the index requires neither block verification, nor modifying block pointers. Shared-to-shared transitions only verify and rewrite blocks in the file referenced by the current write log, which the host processing the write log must have an exclusive lock on.

Unique-to-shared transitions, however, are complicated by the possibility that the file containing the unique block referenced by the index may be locked by some host other than the host processing the write record. While this host may not have access to the indexed block, it does have access to the block referred to by the write log. The host verifies this block’s content and promotes it to a shared block by adding it to the virtual arena and upgrading the index entry accordingly. However, in order to reclaim the originally indexed block, the host must communicate this deduplication opportunity to the host holding the exclusive lock on the file containing the originally indexed block using the associated merge request

file. The host updating the index posts a merge request for the file containing the originally indexed block. This request contains not only the offset of the unique block, but also another COW reference to the shared block. Hosts periodically check for merge requests to the files they have exclusive locks on, verifying any requests they find and merging blocks that pass verification. The COW reference to the shared block in the merge request allows hosts to process requests without accessing the arena.

3.3.3 Garbage Collection

As the host scans the index for hash matches, it also garbage collects unused shared blocks and stale index entries, as shown in Figure 4(d). For each shared block in the index, it checks the file system’s reference count for that block. If the block is no longer in use, it will have only a single reference (from the virtual arena), indicating that it can be removed from the virtual arena and freed. In effect, this implements a simple form of weak references without modifying file system semantics. Furthermore, this approach allows the virtual arena to double as a victim cache before garbage collection has a chance to remove unused blocks.

Unique blocks do not need to be freed, but they can leave behind stale index entries. Hosts garbage collect these by removing any index entries that refer to any block in any of the write records being processed by the host. In the presence of dropped write records, this may not remove all stale index entries, but it will ensure that there is at most one index entry per unique block. In this case, any later write or potential duplicate discovery involving a block with a stale index entry will remove or replace the stale entry. The garbage collection process also check for file truncations and deletions and removes any appropriate index entries.

4 Evaluation

In this section, we present results from the evaluation of our deduplication techniques using various microbenchmarks and realistic workloads. We begin in Section 4.1 with experiments and analysis that shows the space savings achievable with deduplication as well as the space overheads introduced by it, using data from a real corporate VDI deployment. We also draw a comparison against linked clones, an alternative way of achieving space savings.

We have implemented a functional prototype of DEDE atop VMware VMFS. Although we haven’t spent any significant time optimizing it, it is worthwhile examining its basic performance characteristics. In Section 4.2, we present the run-time performance impact of write monitoring and other changes to the file system introduced

by deduplication, as well as the run-time performance gained from improved cache locality. Finally, we look at the performance of the deduplication process itself in Section 4.3.

4.1 Analysis of Virtual Disks in the Wild

To evaluate the usefulness of deduplication in our target workload segment of VDI, we analyzed the virtual disks from a production corporate VDI cluster serving desktop VMs for approximately 400 users on top of a farm of 32 VMware ESX hosts. Out of these, we selected 113 VMs at random to analyze for duplicate blocks, totaling 1.3 TB of data (excluding blocks consisting entirely of NULL bytes). Each user VM belonged exclusively to a single corporate user from a non-technical department like marketing or accounting. The VMs have been in use for six to twelve months and all originated from a small set of standardized Windows XP images. From our experience, this is typical for most enterprise IT organizations, which limit the variation of operating systems to control management and support costs.

Figure 5 shows the reduction in storage space for this VDI farm using deduplication block sizes between 4 KB and 1 MB. As expected, VDI VMs have a high degree of similarity, resulting in an ~80% reduction in storage footprint for the 4 KB block size, which falls off logarithmically to ~35% for 1 MB blocks. Deduplication at the 4 KB block size reduces the original 1.3 TB of data to 235 GB. Given the significant advantage of small block sizes, we chose to use a default 4 KB block size for DEDE. However, a reasonable argument can be made for the smaller metadata storage and caching overhead afforded by an 8 KB block size. We are exploring this as well as dynamic block size selection as future work.

Figure 6 shows a CDF of the same data, detailing the duplication counts of individual blocks in terms of the number of references to each block in the file system *after* deduplication. For example, at the 4 KB block size, 94% of deduplicated blocks are referenced 10 or fewer times by the file system (equivalently, 6% of deduplicated blocks are referenced more than 10 times). Thus, in the original data, most blocks were duplicated a small number of times, but there was a very long tail where some blocks were duplicated many times. At the very peak of the 4 KB distribution, some blocks were duplicated over 100,000 times. Each of these blocks individually represented over 400 MB of space wasted storing duplicate data. Overall, this data serves to show the potential for space savings from deduplication in VDI environments.

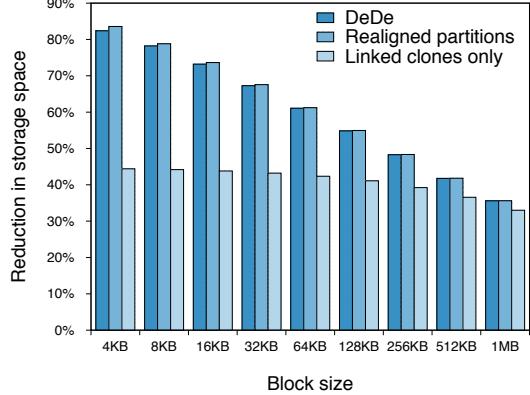


Figure 5: Duplication available at various block sizes and for different variations on the approach. Data is from a production VDI deployment of 113 Windows XP VMs.

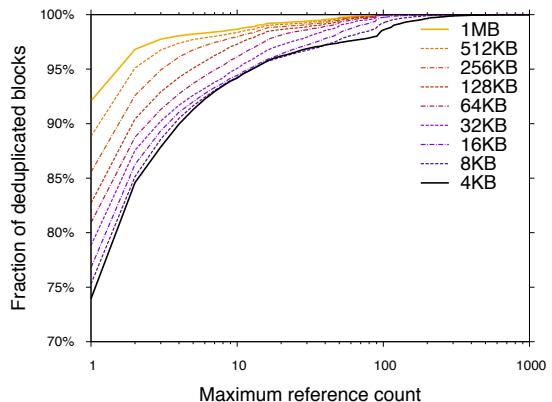


Figure 6: CDF of block duplication counts. A few blocks occur over 100,000 times. Data is from the same deployment as shown in Figure 5.

4.1.1 Space Overheads

While DEDE reduces the amount of space required by file data, it requires additional space for both the index and the additional metadata introduced by mixed block sizes. For our VDI data set, at a 4 KB block size, this additional data totaled 2.7 GB, a mere 1.1% overhead beyond the deduplicated file data.

The index represented 1.5 GB of this overhead, 194 MB of which was file system metadata (pointer blocks) for the virtual arena. The size of the index scales linearly with the size of the deduplicated data because each deduplicated block has one index entry. However, its relative overhead does vary with the ratio of unique to shared blocks, because shared blocks require 4 bytes to locate plus virtual arena metadata, while unique blocks require 12 bytes beyond the 18 bytes required on average for each entry's header and hash. However, even in the worst case, the index represents only 0.73% overhead.

Prior to deduplication, file metadata (inodes and pointer blocks) represented a mere 0.0004% overhead, owing to the efficiency of tracking VMFS’s 1 MB file blocks. After deduplication, each 1 MB block that was divided into sub-blocks requires a new pointer block at 1 KB apiece. As a result, metadata overhead increased to 0.49% after deduplication, or 1.1 GB of data in total. While this is a dramatic increase, metadata is still a very small fraction of the overall space.

4.1.2 Partition Alignment Issues

Our approach of dividing disks into fixed size blocks is sensitive to the alignment of data on those disks. Unfortunately, for historical reasons, the first partition of partition tables created by utilities like `fdisk` on commodity PC systems has a start address 512 bytes short of a 4 KB boundary, which can in turn cause all logical file system blocks to straddle 4 KB disk block boundaries. This has well-known negative performance effects [22], particularly for storage array caches, which are forced to fetch two blocks for each requested file system block. We were initially concerned that this partition misalignment could negatively impact deduplication opportunities, so we “fixed” the alignment of our VDI data by shifting all of the virtual disks by 512 bytes. Figure 5 compares the results of deduplication with and without this realignment and shows that, in practice, partition alignment actually had very *little* impact on achieved deduplication. While this may still prove to be a problem for well-aged guest file systems, if necessary, it can be solved in a virtualized environment by padding the virtual disk image file to realign the guest file system blocks with the host file system blocks.

4.1.3 Deduplication Versus Linked Clones

Linked clones are a simpler space saving alternative to deduplication where individual user VMs are initially constructed as block-level COW snapshots of a golden master VM. This uses the same COW mechanism as DEDE, but all sharing happens during VM creation and the user VM images strictly diverge from the base disk and from each other over time.

In order to compare the efficacy of linked clones versus full deduplication, we simulated the structured sharing of linked clones on our VDI data set. This comparison was necessarily imperfect because we had access to neither the base disks nor ancestry information for the VDI VMs, but it did yield a *lower bound* on the total space required by linked clones. The analysis used our regular deduplication algorithm but restricted it to deduplicating blocks only when they were at the same offset in two files, a reasonable approximation to user disks that are a mini-

%Sequential	Baseline			DEDE		
	T (MB/s)	L (ms)	CPU	T (MB/s)	L (ms)	CPU
100%	233	8.6	33%	233	8.6	220%
0%	84	24	16%	84	24	92%

Table 1: Overhead of in-band write monitoring on a pure IO workload. Results are in terms of throughput (T) and latency (L) for Iometer issuing 32 outstanding 64 KB IOs to a 5 GB virtual disk. The CPU column denotes the utilized processor time relative to a single core.

mal delta from the base disk (*e.g.*, no security patches or software updates have been installed in the user disks).

Figure 5 compares the savings achieved by linked clones against those achieved by DEDE, again at various COW block sizes. Linked clones max out at a 44% reduction in space, reducing the 1.3 TB of original data to 740 GB, a storage requirement over three times larger than full deduplication achieved.

4.2 Run-time Effects of Deduplication

DEDE operates primarily out of band and engenders no slowdowns for accessing blocks that haven’t benefited from deduplication. It can also improve file system performance in certain workloads by reducing the working set size of the storage array cache. For access to deduplicated blocks, however, in-band write monitoring and the effects of COW blocks and mixed block sizes can impact the regular performance of the file system. Unless otherwise noted, all of our measurements of the run-time effects of deduplication were performed using Iometer [9] in a virtual machine stored on a 400 GB 5-disk RAID-5 volume of an EMC CLARiiON CX3-40 storage array.

4.2.1 Overhead of In-Band Write Monitoring

Since DEDE’s design is resilient to dropped write log entries, if the system becomes overloaded, we can shed or defer the work of in-band hash computation based on user-specified policy. Still, if write monitoring is enabled, the hash computation performed by DEDE on every write IO can represent a non-trivial overhead.

To understand the worst-case effect of this, we ran a write-intensive workload with minimal computation on a 5 GB virtual disk. Table 1 shows that these worst case effects can be significant. For example, for a 100% sequential, 100% write workload, the CPU overhead was 6.6× that of normal at the same throughput level. However, because VMware ESX Server offloads the execution of the IO issuing path code, including the hash computation, onto idle processor cores, the actual IO throughput of this workload was unaffected.

	Baseline	Error	SHA-1	Error
Operations/Min	29989	1.4%	29719	0.8%
Response Time (ms)	60 ms	0.8%	61ms	1.4%

Table 2: Overhead of in-band write monitoring on a SQL Server database VM running an online e-commerce application. The mean transaction rate (operations/min) and response times for 10 runs are within noise for this workload. The reported “error” is standard deviation as a percentage of mean.

We don’t expect the effect of the additional computation to be a severe limitation in realistic workloads, which, unlike our microbenchmark, perform computation in addition to IO. To illustrate this, we ran the in-band SHA-1 computation on a realistic enterprise workload. We experimented with a Windows Server 2003 VM running a Microsoft SQL Server 2005 Enterprise Edition database configured with 4 virtual CPUs, 6.4 GB of RAM, a 10 GB system disk, a 250 GB database disk, and a 50 GB log disk. The database virtual disks were hosted on an 800 GB RAID-0 volume with 6 disks; log virtual disks were placed on a 100 GB RAID-0 volume with 10 disks. We used the Dell DVD store (DS2) database test suite [2], which implements a complete online e-commerce application, to stress the SQL database and measure its transactional throughput and latency. The DVD Store workload issues random 8 KB IOs with a write/read ratio of 0.25, and a highly variable number of outstanding write IOs peaking around 28 [7]. Table 2 reports a summary of overall application performance with and without the in-band SHA-1 computation for writes. For this workload, we observed no application-visible performance loss, though extra CPU cycles on other processor cores were being used for the hash computations.

4.2.2 Overhead of COW Specialization

Writing to a COW block in VMFS is an expensive operation, though the current implementation is not well optimized for the COW sub-blocks used extensively by DEDE. In our prototype, it takes \sim 10 ms to specialize a COW block, as this requires copying its content into a newly allocated block in order to update it. As such, any workload phase shift where a large set of previously deduplicated data is being specialized will result in significant performance loss. However, in general, we expect blocks that are identical between VMs are also less likely to be written to and, unlike most approaches to deduplication, we do not suffer this penalty for writes to unique blocks. Optimizations to delay sharing until candidate blocks have been “stable” for some length of time may help further mitigate this overhead, as suggested in [8].

% Sequential	IO Type	Throughput (MB/s)		Overhead
		BS=1 MB	BS=4 KB	
100%	Writes	238	150	37%
	Writes	66	60	9%
100%	Reads	245	135	45%
	Reads	37	32	14%

Table 3: Overhead of mixed block fragmentation. Throughput achieved for 64 KB sequential and random workloads with 16 outstanding IOs. The comparison is between two virtual disks backed by block sizes (BS) of 1 MB and 4 KB, respectively. In the 4 KB case, the virtual disk file consists of 163 disjoint fragments, which implies a sequential run of 31 MB on average.

4.2.3 Overhead of Mixed Block Sizes

VMFS’s 1 MB file blocks permit very low overhead translation from virtual disk IO to operations on the physical disk. While the mixed block size support we added to VMFS is designed to retain this efficiency whenever 1 MB blocks can be used, it unavoidably introduces overhead for 4 KB blocks from traversing the additional pointer block level and increased external fragmentation.

To measure the effects of this, we compared IO to two 5 GB virtual disks, one backed entirely by 1 MB blocks and one backed entirely by 4 KB blocks. These configurations represent the two extremes of deduplication: all unique blocks and all shared blocks, respectively. The first disk required one pointer block level and was broken into 3 separate extents on the physical disk, while the second disk required two pointer block levels and spanned 163 separate extents.

The results of reading from these virtual disks are summarized in Table 3. Unfortunately, sub-blocks introduced a non-trivial overhead for sequential IO. This is partly because VMFS’s sub-block placement and IO handling is not yet well-optimized since sub-blocks have not previously been used in the VM IO critical path, whereas VMFS’s file block IO has been heavily optimized. One possible way to mitigate this overhead is by preventing the deduplication process from subdividing file blocks unless they contain some minimum number of 4 KB candidates for sharing. This would impact the space savings of deduplication, but would prevent DEDE from subdividing entire file blocks for the sake of just one or two sharable blocks. Improvements in sub-block IO performance and block subdivision are considered future work.

4.2.4 Disk Array Caching Benefits

For some workloads, deduplication can actually *improve* run-time performance by decreasing the storage array cache footprint of the workload. To demonstrate this, we

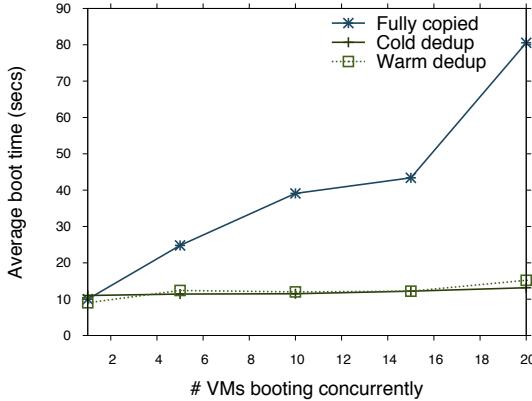


Figure 7: Windows XP VM boot up time comparison between fully copied VMs and deduplicated VMs. Deduplicated VMs are booted twice in order to measure the impact of writing to deduplicated blocks.

picked a common, critical, time-limited VDI workload: booting many VMs concurrently. VDI boot storms can happen as part of a nightly cycle of shutting down VMs and their hosts to conserve power, from patching guest operating systems *en masse*, from cluster fail-over, or for a myriad of other reasons.

To test the cache effects of deduplication, we compared the average time required to boot from one to twenty VMs simultaneously between two configurations: (1) the VMs were each full copies of the golden VM (much like the VDI configuration from Section 4.1) and (2) VMs were deduplicated copies. The results plotted in Figure 7 show a dramatic improvement of deduplication versus full copies, owing to the decrease in cache footprint.

To further validate the overhead of COW specialization for a realistic workload, we also booted the set of VMs a second time after deduplication. The disk images were “cold” the first time; they consisted entirely of COW blocks. The second time, any blocks written to were already specialized and could be written to directly. The graph shows virtually no difference between these two cases, indicating that COW specialization overhead is not an issue for this workload. This is not unexpected, as there are only a few write operations during VM boot.

4.3 Deduplication Rate

While our prototype’s implementation of indexing has not yet been optimized, we measured the overall rate at which it could process modified blocks, as well as the performance of the three main operations performed by it: scanning the index, subdividing 1 MB blocks into 4 KB blocks, and COW sharing duplicates.

The index scanning process operates at nearly the disk’s sequential access rate, as discussed in Section 3.2.2.

At \sim 23 bytes per index entry, our prototype can process entries for 6.6 GB of blocks *per second*. However, unlike block subdivision and COW sharing, which require time proportional to the number of newly shared blocks, the index scan requires time proportional to the total number of blocks in the file system, so it is critical that this be fast. Once new duplicates have been discovered by the index scan, 1 MB file blocks containing any of these duplicates can be subdivided into 4 KB blocks at 37.5 MB/sec. Finally, these newly discovered duplicates can be eliminated via COW sharing at 2.6 MB/sec.

The COW sharing step limits our prototype to processing \sim 9 GB of new *shared* blocks per hour. Unique blocks (*i.e.*, recently modified blocks whose hashes do not match anything in the index) can be processed at the full index scan rate. Furthermore, provisioning from templates, a source of large amounts of duplicate data, can be performed directly as a COW copy (at roughly 1 GB/sec), so our deduplication rate applies only to duplicates that arise outside of provisioning operations. Still, we feel that our COW sharing rate can be significantly improved with more profiling and optimization effort. However, even at its current rate, the prototype can eliminate duplicates at a reasonable rate for a VDI workload given only a few off-peak hours per day to perform out of band deduplication.

5 Related Work

Much work has been done towards investigating deduplication for file systems with a centralized component. Venti [16] pioneered the application of content-addressable storage (CAS) to file systems. Venti is a block storage system in which blocks are identified by a collision-resistant cryptographic hash of their contents and stored in an append-only log on disk. An on-disk index structure maps from content hashes to block locations. Venti’s append-only structure makes it well suited to archival, but not to live file systems. Venti also depends heavily on a central server to maintain the block index.

Various other systems, notably Data Domain’s archival system [26] and Foundation [17], have extended and enhanced the Venti approach, but still follow the same basic principles. While deduplication for archival is generally well understood, deduplication in live file systems presents very different challenges. Because backup systems are concerned with keeping data for arbitrarily long periods of time, backup deduplication can rely on relatively simple append-only data stores. Data structures for live deduplication, however, must be amenable to dynamic allocation and garbage collection. Furthermore, live file systems, unlike backup systems, are latency sensitive for both reading and writing. Thus, live file system deduplication must have minimal impact on these criti-

cal paths. Backup data also tends to be well-structured and presented to the backup system in sequential streams, whereas live file systems must cope with random writes.

Many CAS-based storage systems, including [5,16,20], address data exclusively by its content hash. Write operations return a content hash which is used for subsequent read operations. Applying this approach to VM disk storage implies multi-stage block address resolution, which can negatively affect performance [10]. Furthermore, since data is stored in hash space, spatial locality of VM disk data is lost, which can result in significant loss of performance for some workloads. DEDE avoids both of these issues by relying on regular file system layout policy and addressing all blocks by \langle filename, offset \rangle tuples, rather than content addresses. DEDE uses content hashes only for identifying duplicates.

Both NetApp’s ASIS [14] and Microsoft’s Single Instance Store [1] use out of band deduplication to detect duplicates in live file systems in the background, similar to DEDE. SIS builds atop NTFS and applies content-addressable storage to whole files, using NTFS filters to implement file-level COW-like semantics.

While SIS depends on a centralized file system and a single host to perform scanning and indexing, Farsite builds atop SIS to perform deduplication in a distributed file system [3]. Farsite assigns responsibility for each file to a host based on a hash of the file’s content. Each host stores files in its local file system, relying on SIS to locally deduplicate them. However, this approach incurs significant network overheads because most file system operations, including reads, require cross-host communication and file modifications require at least updating the distributed content hash index.

Hong’s Duplicate Data Elimination (DDE) system [8] avoids much of the cross-host communication overhead of Farsite by building from IBM’s Storage Tank SAN file system [11]. DDE hosts have direct access to the shared disk and can thus read directly from the file system. However, metadata operations, including updates to deduplicated shared blocks, must be reported to a centralized metadata server, which is solely responsible for detecting and coalescing duplicates. DEDE is closest in spirit to DDE. However, because DEDE uses a completely decentralized scheme with no metadata server, it doesn’t suffer from single points of failure or contention. Furthermore, DEDE prevents cross-host concurrency issues by partitioning work and relying on coarse-grain file locks, whereas DDE’s approach of deduplicating from a central host in the midst of a multi-host file system introduces complex concurrency issues.

Numerous studies have addressed the effectiveness of content-addressable storage for various workloads. Work that has focused on VM deployments [12, 17] has concluded that CAS was very effective at reducing storage

space and network bandwidth compared to traditional data reduction techniques like compression.

Other work has addressed deduplication outside of file systems. Our work derives inspiration from Waldspurger [25] who proposed deduplication of memory contents, now implemented in the VMware ESX Server hypervisor [23]. In this system, identical memory pages from multiple virtual machine are backed by the same page and marked copy-on-write. The use of sharing hints from that work is analogous to our merge requests.

6 Conclusion

In this paper, we studied deduplication in the context of decentralized cluster file systems. We have described a novel software system, DEDE, which provides block-level deduplication of a live, shared file system without any central coordination. Furthermore, DEDE builds atop an existing file system without violating the file system’s abstractions, allowing it to take advantage of regular file system block layout policies and in-place updates to unique data. Using our prototype implementation, we demonstrated that this approach can achieve up to 80% space reduction with minor performance overhead on realistic workloads.

We believe our techniques are applicable beyond virtual machine storage and plan to examine DEDE in other settings in the future. We also plan to explore alternate indexing schemes that allow for greater control of deduplication policy. For example, high-frequency deduplication could prevent temporary file system bloat during operations that produce large amounts of duplicate data (*e.g.*, mass software updates), and deferral of merge operations could help reduce file system fragmentation. Additionally, we plan to further explore the trade-offs mentioned in this paper, such as block size versus metadata overhead, in-band versus out-of-band hashing, and sequential versus random index updates.

DEDE represents just one of the many applications of deduplication to virtual machine environments. We believe that the next step for deduplication is to integrate and unify its application to file systems, memory compression, network bandwidth optimization, etc., to achieve end-to-end space and performance optimization.

Acknowledgments

We would like to thank Mike Nelson, Abhishek Rai, Manjunath Rajashekhar, Mayank Rawat, Dan Scales, Dragan Stancevic, Yuen-Lin Tan, Satyam Vaghani, and Krishna Yadappanavar, who, along with two of the coauthors, developed the core of VMFS in unpublished work, which this paper builds on top of. We are thankful to Orran

Krieger, James Cipar, and Saman Amarasinghe for conversations that helped clarify requirements of an online deduplication system. We are indebted to our shepherd Andrew Warfield, the anonymous reviewers, John Blumenthal, Mike Brown, Jim Chow, Peng Dai, Ajay Gulati, Jacob Henson, Beng-Hong Lim, Dan Ports, Carl Waldspurger and Xiaoyun Zhu for providing detailed reviews of our work and their support and encouragement. Finally, thanks to everyone who has noticed the duplication in our project codename and brought it to our attention.

This material is partly based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

- [1] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows®2000. In *Proceedings of the 4th USENIX Windows Systems Symposium (WSS '00)*, Seattle, WA, Aug. 2000. USENIX.
- [2] Dell, Inc. DVD Store. <http://delltechcenter.com/page/DVD+store>.
- [3] J. Douceur, A. Adya, W. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, Vienna, Austria, July 2002. IEEE.
- [4] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)*, San Francisco, CA, Feb. 2009. USENIX.
- [5] EMC Centera datasheet. <http://www.emc.com/products/detail/hardware/centera.htm>.
- [6] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3), Sept. 1979.
- [7] A. Gulati, C. Kumar, and I. Ahmad. Storage workload characterization and consolidation in virtualized environments. In *2nd International Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.
- [8] B. Hong, D. Plantenberg, D. D. E. Long, and M. Sivan-Zimet. Duplicate data elimination in a SAN file system. In *Proceedings of the 21st Symposium on Mass Storage Systems (MSS '04)*, Goddard, MD, Apr. 2004. IEEE.
- [9] Iometer. <http://www.iometer.org/>.
- [10] A. Liguori and E. V. Hensbergen. Experiences with content addressable storage and virtual disks. In *Proceedings of the Workshop on I/O Virtualization (WIOV '08)*, San Diego, CA, Dec. 2008. USENIX.
- [11] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM storage tank—a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2), 2003.
- [12] P. Nath, M. A. Kozuch, D. R. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATEC '06)*, Boston, MA, June 2006. USENIX.
- [13] P. Nath, B. Urgaonkar, and A. Sivasubramaniam. Evaluating the usefulness of content addressable storage for high-performance data intensive applications. In *Proceedings of the 17th High Performance Distributed Computing (HPDC '08)*, Boston, MA, June 2008. ACM.
- [14] Netapp Deduplication (ASIS). <http://www.netapp.com/us/products/platform-os/dedupe.html>.
- [15] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proceedings of the 16th Symposium on Mass Storage Systems (MSS '99)*, San Diego, CA, Mar. 1999. IEEE.
- [16] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)* [19].
- [17] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in Foundation. In *Proceedings of the USENIX Annual Technical Conference (ATEC '08)*, Boston, MA, June 2008. USENIX.
- [18] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)* [19].
- [19] USENIX. *The 1st USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, Jan. 2002.
- [20] M. Vilayannur, P. Nath, and A. Sivasubramaniam. Providing tunable consistency for a parallel file store. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, CA, Dec. 2005. USENIX.
- [21] VMware, Inc. VMFS datasheet. http://www.vmware.com/pdf/vmfs_datasheet.pdf.
- [22] VMware, Inc. Recommendations for aligning VMFS partitions. Technical report, Aug. 2006.
- [23] VMware, Inc. *Introduction to VMware Infrastructure*. 2007. <http://www.vmware.com/support/pubs/>.
- [24] VMware, Inc. VMware Virtual Desktop Infrastructure (VDI) datasheet, 2008. http://www.vmware.com/files/pdf/vdi_datasheet.pdf.
- [25] C. A. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec. 2002. USENIX.
- [26] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, Feb. 2008. USENIX.

FlexFS: A Flexible Flash File System for MLC NAND Flash Memory

Sungjin Lee[†], Keonsoo Ha[†], Kangwon Zhang[†], Jihong Kim[†], and Junghwan Kim^{*}

[†]*Seoul National University, Korea*

{chamdo, air21c, kwzhang, jihong}@davinci.snu.ac.kr

^{*}*Samsung Electronics, Korea*

junghwani.kim@samsung.com

Abstract

The multi-level cell (MLC) NAND flash memory technology enables multiple bits of information to be stored on a single cell, thus making it possible to increase the density of the memory without increasing the die size. For most MLC flash memories, each cell can be programmed as a single-level cell or a multi-level cell during runtime. Therefore, it has a potential to achieve both the high performance of SLC flash memory and the high capacity of MLC flash memory.

In this paper, we present a flexible flash file system, called FlexFS, which takes advantage of the dynamic re-configuration facility of MLC flash memory. FlexFS divides the flash memory medium into SLC and MLC regions, and dynamically changes the size of each region to meet the changing requirements of applications. We exploit patterns of storage usage to minimize the overhead of reorganizing two different regions. We also propose a novel wear management scheme which mitigates the effect of the extra writes required by FlexFS on the lifetime of flash memory. Our implementation of FlexFS in the Linux 2.6 kernel shows that it can achieve a performance comparable to SLC flash memory while keeping the capacity of MLC flash memory for both simulated and real mobile workloads.

1 Introduction

As flash memory technologies quickly improve, NAND flash memory is becoming an attractive storage solution for various IT applications from mobile consumer electronics to high-end server systems. This rapid growth is largely driven by the desirable characteristics of NAND flash memory, which include high performance and low-power consumption.

There are two types of NAND flash memory in the market: a single-level cell (SLC) and a multi-level cell (MLC) flash memory. They are distinctive in terms of

capacity, performance, and endurance. The capacity of MLC flash memory is larger than that of SLC flash memory. By storing two (or more) bits on a single memory cell, MLC flash memory achieves significant density increases while lowering the cost per bit over SLC flash memory which can only store a single bit on a cell. However, SLC flash memory has a higher performance and a longer cell endurance over MLC flash memory. Especially, the write performance of SLC flash memory is much higher than that of MLC flash memory.

As the demand for the high capacity storage system is rapidly increasing, MLC flash memory is being widely adopted in many mobile embedded devices, such as smart phones, digital cameras, and PDAs. However, because of a poor performance characteristic of MLC flash memory, it is becoming harder to satisfy users' requirements for the high performance storage system while providing increased storage capacity.

To overcome this poor performance, in this paper, we propose exploiting the flexible programming feature of MLC flash memory [1]. Flexible programming is a writing method which enables each cell to be programmed as a single-level cell (SLC programming) or a multi-level cell (MLC programming). If SLC programming is used to write data into a particular cell, the effective properties of that cell become similar to those of an SLC flash memory cell. Conversely, MLC programming allows us to make use of the high capacity associated with MLC flash memory.

The most attractive aspect of flexible programming is that it allows fine-grained storage optimizations, in terms of both performance and capacity, to meet the requirements of applications. For instance, if the current capacity of flash memory is insufficient for some application, MLC flash memory can change its organization and increase the number of multi-level cells to meet the space requirement. However, to exploit flexible cell programming effectively, several issues need to be considered.

First, heterogeneous memory cells should be managed

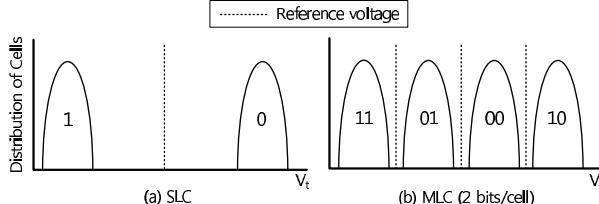


Figure 1: Threshold voltage distributions for SLC (1 bit/cell) and MLC (2 bits/cell)

in a way that is transparent to the application layer, because flexible programming allows two different types of a cell to exist in the same flash chip simultaneously.

Second, dynamic cell reconfigurations between the SLC and MLC must be handled properly. For example, if too many flash cells are used as single-level cells, the capacity of flash memory might be critically impaired, even though the overall I/O performance is improved. Therefore, it is important to determine the number of SLC cells and MLC cells so that both the performance and capacity would be optimally supported.

Third, the cost of dynamic cell reconfigurations should be kept as low as possible. Changing the type of a cell requires expensive erase operations. Since an erase operation resets cells to their initial bit value (e.g., 1), the data stored in the cells must first be moved to elsewhere. The performance overhead of this data migration impairs the overall I/O performance.

Finally, write and erase operations required to change the type of a cell reduce the endurance of each cell, resulting in the decrease of the lifetime of flash memory. This problem also needs to be addressed properly.

In this paper, we propose a flexible flash file system, called *FlexFS*, for MLC flash memory that addresses the above requirements effectively. FlexFS provides applications with a homogeneous view of storage, while internally managing two heterogeneous memory regions, an SLC region and an MLC region. FlexFS guarantees the maximum capacity of MLC flash memory to users while it tries to write as much data as possible to the SLC region so as to achieve the highest I/O performance. FlexFS uses a data migration policy to compensate for the reduced capacity caused by overuse of the SLC region. In order to prolong the lifespan of flash memory, a new wear management scheme is also proposed.

In order to evaluate the effectiveness of FlexFS, we implemented FlexFS in the Linux 2.6.15 kernel on a development board. Evaluations were performed using synthetic and real workloads. Experimental results show that FlexFS achieves 90% of the read and 96% of the write performance of SLC flash memory, respectively, while offering the capacity of MLC flash memory.

The rest of this paper is organized as follows. In Sec-

tion 2, we present a brief review of NAND flash memory and explain MLC flash memory in detail. In Section 3, we give an overview of FlexFS and introduce the problems that occur with a naive approach to exploiting flexible cell programming. In Section 4, we describe SLC and MLC management techniques. In Section 5, we present experimental results. Section 6 describes related work on heterogeneous storage systems. Finally, in Section 7, we conclude with a summary and future work.

2 Background

2.1 NAND Flash Memory

NAND flash memory consists of multiple blocks, each of which is composed of several pages. In many NAND flash memories, the size of a page is between 512 B and 4 KB, and one block consists of between 4 and 128 pages. NAND flash memory does not support an overwrite operation because of its write-once nature. Therefore, before writing new data into a block, the previous data must be erased. Furthermore, the total number of erasures allowed for each block is typically limited to between 10,000 and 100,000 cycles.

Like SRAM and DRAM, flash memory stores bits in a memory cell, which consists of a transistor with a floating gate that can store electrons. The number of electrons stored on the floating gate determines the threshold voltage, V_t , and this threshold voltage represents the state of the cell. In case of a single-level cell (SLC) flash memory, each cell has two states, and therefore only a single bit can be stored in that cell. Figure 1(a) shows how the value of a bit is determined by the threshold voltage. If the threshold voltage is greater than a reference voltage, it is interpreted as a logical ‘1’; otherwise, it is regarded as a logical ‘0’. In general, the write operation moves the state of a cell from ‘1’ to ‘0’, while the erase operation changes ‘0’ to ‘1’.

If flash memory is composed of memory cells which have more than two states, it is called a multi-level cell (MLC) flash memory, and two or more bits of information can be stored on each cell, as shown in Figure 1(b). Even though the density of MLC flash memory is higher than that of SLC flash memory, it requires more precise charge placement and charge sensing (because of narrower voltage ranges for each cell state), which in turn reduces the performance and endurance of MLC flash memory in comparison to SLC flash memory.

2.2 MLC NAND Flash Memory Array

In MLC flash memory, it is possible to use SLC programming, allowing a multi-level cell to be used as a single-level cell. To understand the implications of SLC

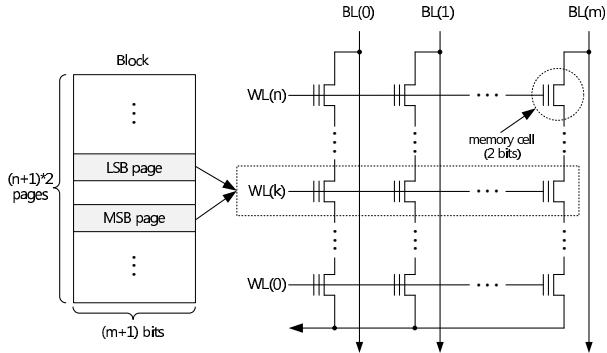


Figure 2: An organization of an MLC flash memory array (2 bits/cell)

programming, it is necessary to know the overall architecture of a flash memory array. Figure 2 illustrates the array of flash memory cells which forms a flash memory block. We assume that each cell is capable of holding two bits. For a description purpose, this figure does not show all the elements, such as source and drain select gates, which are required in a memory array. (For a more detailed description, see references [2, 3].)

As shown in Figure 2, the memory cells are arranged in an array of rows and columns. The cells in each row are connected to a word line (e.g., $WL(0)$), while the cells in each column are coupled to a bit line (e.g., $BL(0)$). These word and bit lines are used for read and write operations. During a write operation, the data to be written ('1' or '0') is provided at the bit line while the word line is asserted. During a read operation, the word line is again asserted, and the threshold voltage of each cell can then be acquired from the bit line.

Figure 2 also shows the conceptual structure of a flash block corresponding to a flash memory array. The size of a page is determined by the number of bit lines in the memory array, while the number of pages in each flash block is twice the number of word lines, because two different pages share the memory cells that belong to the same word line. These two pages are respectively called the least significant bit (LSB) page and the most significant bit (MSB) page. As these names imply, each page only uses its own bit position of a bit pattern stored in a cell. (This is possible because each memory cell stores two bits, for example, one bit for the LSB page and the other for the MSB page.) Thus, if a block has 128 pages, there are 64 LSB and 64 MSB pages.

Because multiple pages are mapped to the same word line, read and write operations must distinguish the destination page of each operation. For example, if a cell is in an erased state (i.e., a logical '11') and a logical '0' is programmed to the MSB position of the cell, the cell will then have a bit pattern of '01', which is interpreted as a

Table 1: Performance comparison of different types of cell programming (us)

Operation	<i>SLC</i>	<i>MLC_{LSB}</i>	<i>MLC_{BOTH}</i>
Read (page)	399	409	403
Write (page)	417	431	994
Erase (block)	860	872	872

logical '0' for the MSB page. If the LSB position is then programmed as '0', the bit pattern will change to '00'.

2.3 SLC Programming in MLC

Since MLC flash memory stores multiple pages in the same word line, it is possible for it to act as SLC flash memory by using only the LSB pages (or MSB pages, depending on the manufacturer's specification). Thus, SLC programming is achieved by only writing data to the LSB pages in a block. In this case, since only two states of a cell, '11' and '10', are used shown in Figure 1(b), the characteristics of a multi-level cell become very similar to those of a single-level cell. The logical offsets of the LSB and MSB pages in a block are determined by the flash memory specification, and therefore SLC programming can be managed at the file system level. Naturally, SLC programming reduces the capacity of a block by half, because only the LSB pages can be used.

Table 1 compares the performance of the three different types of cell programming method. The *SLC* column shows the performance data in a pure SLC flash memory; the *MLC_{LSB}* column gives the performance data when only the LSB pages are used; and the *MLC_{BOTH}* column gives the data when both the LSB and MSB pages are used. The access times for page reads and writes, and for block erase operations were measured using the Samsung's KFXXGH6X4M flash memory [4] at the device driver interface level. As shown in Table 1, there are no significant performance differences between page read and block erase operations for the three programming methods. However, the write performance is significantly improved with *MLC_{LSB}*, and approaches to that of *SLC*.

This improvement in the write performance under *MLC_{LSB}* is the main motivation for FlexFS. Our primary goal is to improve the write performance of MLC flash memory using the *MLC_{LSB}* method, while maintaining the capacity of MLC flash memory using the *MLC_{BOTH}* method.

3 Overview of the FlexFS File System

We will now describe the overall architecture of the proposed FlexFS system. FlexFS is based on JFFS2 file sys-

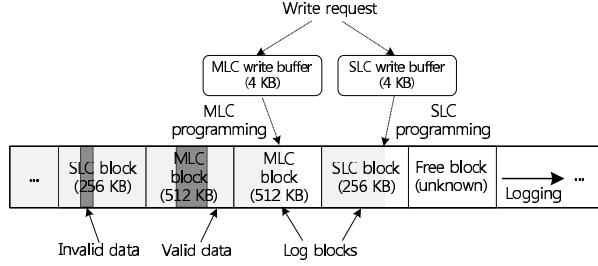


Figure 3: The layout of flash blocks in FlexFS

tem [5], and hence the overall architecture is very similar to JFFS2 except for some features required to manage heterogeneous cells and to exploit flexible programming. Therefore, in this section, we focus on how FlexFS deals with different types of a cell. We also introduce a baseline approach to exploit flexible cell programming in order to illustrate the need for better policies, which will be introduced in detail on the following section.

3.1 Design Overview

In order to manage heterogeneous cells efficiently, FlexFS logically divides the flash memory medium into an SLC region, composed of SLC blocks, and an MLC region consisting of MLC blocks. If a block does not contain any data, it is called a free block. In FlexFS, a free block is neither an SLC block nor an MLC block; its type is only determined when data is written into it.

Figure 3 shows the layout of flash memory blocks in FlexFS. We assume that the number of pages in a block is 128, and the page size is 4 KB. (These values will be used throughout the rest of this paper.) When a write request arrives, FlexFS determines the type of region to which the data is to be written, and then stores the data temporarily in an appropriate write buffer. This temporary buffering is necessary because the unit of I/O operations is a single page in flash memory. Therefore, the write buffer stores the incoming data until there is at least the page size of data (i.e., 4 KB), which can be transferred to flash memory. In order to ensure the data reliability, if there is an explicit flush command from the operating system, all the pending data is immediately written to flash memory. In FlexFS, separate write buffers are used for the SLC and MLC regions.

FlexFS manages flash memory in a similar fashion to other log-structured file systems [5, 6, 7], except that two log blocks (one for the SLC and another for the MLC region) are reserved for writing. When data is evicted from the write buffer to flash memory, FlexFS writes them sequentially from the first page to the last page of the corresponding region's log block. MLC programming is used to write data to the MLC block, and SLC programming

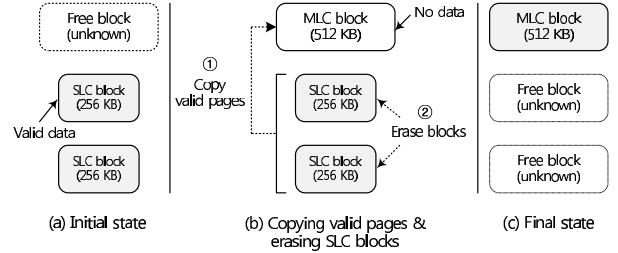


Figure 4: Steps in data migration

is used to write to the SLC block. If existing data is updated, the old version of the data is first invalidated, while the new data is appended to the free space of a log block. The space used by this invalid data is later reclaimed by the garbage collector (Section 4.3).

After all the free pages in the current log block have been exhausted, a new log block is allocated from the free blocks. However, if there is not enough free space to store the data, the data migrator triggers a data migration (Section 4.1.1) to create more free space. This expands the effective capacity of flash memory by moving the data from the SLC region to the MLC region. Figure 4 illustrates the steps in data migration. In this example, there are initially two SLC blocks and one free block, as shown in Figure 4(a). We assume that all the pages in the two SLC blocks contain valid data. During the data migration, the free block is converted into an MLC block, and the 128 pages in the two SLC blocks are copied to this MLC block. Then the two SLC blocks are erased, making them free blocks. This migration frees up one block, doubling the remaining capacity of flash memory, as shown in Figure 4(c).

When a read request arrives, FlexFS first checks whether the write buffers contain the requested data. If so, the data in the write buffer is transferred to the page cache. Otherwise, FlexFS searches an inode cache, which is kept in main memory, to find a physical address for the requested file data. The inode cache maintains the inode numbers and physical locations of data that belong to each inode. If the physical address of the required data is found, regardless of the type of block in which the data is stored, FlexFS can read the data from that address.

3.2 Baseline Approach and Its Problems

The major objective of FlexFS is to support both high performance and high capacity in MLC flash memory. A simplistic solution, which we call the baseline approach, is first to write as much data as possible into SLC blocks to maximize the I/O performance. When there are no more SLC blocks available, the baseline approach initiates a data migration so that more space becomes avail-

able for subsequent write requests, so as to maximize the capacity of flash memory. This simple approach has two serious drawbacks.

First, if the amount of data stored on flash memory approaches to half of its maximum capacity, almost all the free blocks are exhausted. This is because the capacity of the SLC block is half that of the MLC block. At this point, a data migration has to be triggered to free some blocks before writing the requested data. But, this reduces the overall I/O performance significantly. To address this problem, we introduce techniques to reduce the migration penalty, or to hide it from users.

Second, the baseline approach degrades the lifetime of MLC flash memory seriously. Each block of NAND flash memory has a finite number of erase cycles before it becomes unusable. The baseline approach tends to increase the number of erase operations because of the excessive data migration. In the worst case, the number of erasures could be three times more than in conventional flash file systems. We solve this problem by controlling the degree of the migration overhead, with the aim of meeting a given lifetime requirement.

4 Design and Implementation of FlexFS

4.1 Reducing the Migration Overhead

To reduce or hide the overhead associated with data migrations, we introduce three techniques: *background migration*, *dynamic allocation*, and *locality-aware data management*. The background migration technique exploits the times when the system is idle to hide the data migration overhead. This technique is effective for many mobile embedded systems (e.g., mobile phones) which have long idle time. The dynamic allocation technique, on the other hand, is aimed at systems with less idle time. By redirecting part of the incoming data into the MLC region depending on the idleness of the system, it reduces the amount of data that is written into the SLC region, which in turn reduces the data migration overheads. The third technique, locality-aware data management, exploits the locality of I/O accesses to improve the efficiency of data migration. We will now look at these three techniques in more detail.

4.1.1 Background Migration Technique

Figure 5 shows the overall process of the background migration. In this figure, the X-axis shows the time and the Y-axis gives the type of job being performed by the file system. A foreground job represents I/O requests issued by applications or the operating system. T_{busy} is a time interval during which the file system is too busy to process foreground jobs, and T_{idle} is an idle interval.

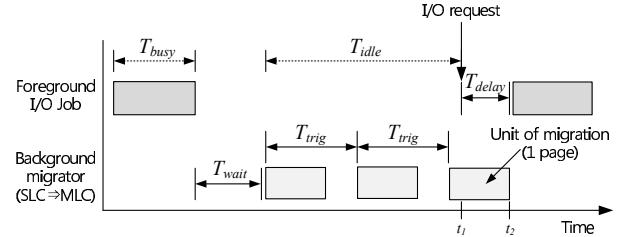


Figure 5: Overview of the background migration

During this idle time the background migrator can move data from the SLC region to the MLC region, thus freeing many blocks. These free blocks can then be used as SLC blocks to store data, and so we can avoid a compulsory data migration if there is sufficient idle time.

In designing the background migration technique, there are two important issues: First, it is important to minimize the delay in response time T_{delay} inflicted on foreground tasks by the background migration. For example, in Figure 5, an I/O request arrives at t_1 , but it cannot proceed until t_2 because of interference from the background migration. So T_{delay} is $t_2 - t_1$. To reduce this delay, the data migrator monitors the I/O subsystem, and suspends the background migration process if there is an I/O request. Since the unit of a data migration is a single page, the maximum delay in response time will be less than the time required to move a page from SLC to MLC (about 1,403 us) theoretically. In addition, we also design the background migrator so that it does not utilize all available idle times. Instead, it periodically invokes a data migration at a predefined triggering interval T_{trig} . If T_{trig} is larger than the time required to move a single page, FlexFS reduces the probability that a foreground job will be issued while a data migration is running, thus further reducing T_{delay} .

The second issue is when to initiate a background migration. Our approach is based on a threshold; if the duration of the idle period is longer than a specific threshold value T_{wait} , then the background migrator is triggered. This kind of problem has been extensively studied in dynamic power management (DPM) of hard disk drives [8], which puts a disk into a low-power state after a certain idle time in order to save energy. However, the transition to a low-power state has to be made carefully because it introduces a large performance penalty. Fortunately, because T_{delay} is quite short, more aggressive transitioning is possible in our background migration technique, allowing T_{wait} to be set to a small value.

4.1.2 Dynamic Allocation Technique

The background migration technique works well when a system has sufficient idle time. Otherwise, the migration

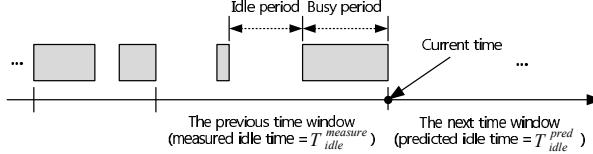


Figure 6: Our approach to idle time prediction

overhead cannot be avoided. But it can be ameliorated by writing part of the incoming data into the MLC region, so as to reduce the amount of data to be moved by the background migrator. Although this approach results in a lower I/O performance than SLC flash memory, it can prevent significant performance degradation due to a compulsory data migration.

The dynamic allocator determines the amount of data that will be written into the SLC region. Intuitively, it is clear that this must depend on how much idle time there is in a given system. Since the amount of idle time changes dynamically with user activities, we need to predict it carefully. Figure 6 illustrates the basic idea of our idle time prediction approach, which is based on previous work [9]. In this figure, each time window represents the period during which N_p pages are written into flash memory. The dynamic allocator stores measured idle times for several previous time windows, and uses them to predict the idle time, T_{idle}^{pred} , for the next time window. The value of T_{idle}^{pred} is a weighted average of the idle times for the latest 10 time windows; the three most recent windows are given a higher weight to take the recency of I/O pattern into account.

If we know the value of T_{idle}^{pred} , we can use it to calculate an allocation ratio, denoted by α , which determines how many pages will be written to the SLC region in the next time window. The value of α can be expressed as follows:

$$\alpha = \begin{cases} 1 & \text{if } T_{idle}^{pred} \geq T_{mig} \\ \frac{T_{idle}^{pred}}{T_{mig}} & \text{if } T_{idle}^{pred} < T_{mig}, \end{cases} \quad (1)$$

$$\text{where } T_{mig} = N_p \cdot (T_{trig} + T_{erase}^{SLC}/S_p^{SLC}), \quad (2)$$

where T_{erase}^{SLC} is the time required to erase an SLC flash block which contains S_p^{SLC} pages. As mentioned in Section 4.1.1, T_{trig} is the time interval required for one page to migrate from the SLC region to the MLC region. Therefore, T_{mig} is the migration time, which includes the time taken to move all N_p pages to the MLC region and the time for erasing all used SLC blocks. If $T_{idle}^{pred} \geq T_{mig}$, there is sufficient idle time for data migrations, and thus $\alpha = 1$. Otherwise, the value of α should be reduced so that less data is written into the SLC region, as expressed by Eq. (1).

Once the value of α has been determined, the dynamic allocator tries to distribute the incoming data across the

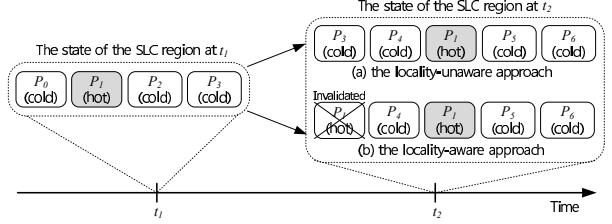


Figure 7: A comparison of the locality-unaware and locality-aware approaches

different flash regions depending on α . Therefore, the number of pages to be written into the SLC region, N_p^{SLC} , and the amount of data destined for the MLC region, N_p^{MLC} , can be expressed as follows:

$$N_p^{SLC} = N_p \cdot \alpha, \quad N_p^{MLC} = N_p \cdot (1 - \alpha). \quad (3)$$

Finally, after writing all N_p pages, the dynamic allocator calculates a new value of α for the next N_p pages.

4.1.3 Locality-aware Data Management Technique

FlexFS is based on a log-structured file system, and therefore it uses the out-place update policy. Under this policy, hot data with a high update frequency generates more outdated versions of itself than cold data, which is updated infrequently. Our locality-aware data management technique exploits this characteristic to increase the efficiency of data migration.

Figure 7 compares the locality-aware and the locality-unaware approaches. We assume that, at time t_1 , three cold pages p_0 , p_2 , and p_3 , and one hot page p_1 , exist in the SLC region. Between t_1 and t_2 , there are some idle periods, and new pages p_4 , p_5 , and p_6 are written into the SLC region. Note that p_1 is rewritten because it contains hot data. In the case of the locality-unaware approach shown in Figure 7(a), we assume that pages p_0 , p_1 , and p_2 are moved to the MLC region during idle time, but p_3 cannot be moved because there is not enough idle time. Therefore, at time t_2 , there are five pages in the SLC region. If the value of N_p is 4, the value of α should decrease so that data will not accumulate in the SLC region. However, if we consider the locality of the data, we can move p_3 instead of p_1 during idle periods, as shown in Figure 7(b). Since p_1 has a high locality, it is highly likely to be invalidated by t_2 . Therefore, an unnecessary page migration for p_1 can be avoided, and only four pages remain in the SLC region. In this case, we need not to reduce the value of α , and more data will be written into the SLC region.

Using this observation, Eq. (2) can be rewritten as follows:

$$T_{mig} = (N_p - N_p^{hot}) \cdot (T_{trig} + T_{erase}^{SLC}/S_p^{SLC}), \quad (4)$$

where N_p^{hot} is the number of page writes for hot pages stored in the SLC region. For instance, in the above example, N_p^{hot} is 1. Because we only need to move $N_p - N_p^{hot}$ pages into the MLC region, the value of T_{mig} can be reduced, allowing an increase in α for the same amount of idle time.

To exploit the locality of I/O references, there are two questions to answer. The first is to determine the locality of a given data. To know the hotness of data, FlexFS uses a 2Q-based locality detection technique [10], which is widely used in the Linux operating system. This technique maintains a hot and a cold queue, each containing a number of nodes. Each node contains the inode number of a file. Nodes corresponding to frequently accessed files are stored on the hot queue, and the cold queue contains nodes for infrequently accessed files. The locality of a given file can easily be determined from queue in which the corresponding node is located.

Second, the data migrator and the dynamic allocator should be modified so that they take the locality of data into account. The data migrator tries to select an SLC block containing cold data as a victim, and an SLC block containing hot data is not selected as a victim unless very few free blocks remain. Since a single block can contain multiple files which have different hotness, FlexFS calculates the average hotness of each block as the criterion, and chooses a block whose hotness is lower than the middle. It seems better to choose a block containing only cold pages as a victim block; if there are only a few bytes of hot data in a victim, this results in useless data migrations for hot data. However, this approach incurs the delay in reclaiming free blocks, because even if the small amount of hot data is stored on a block, the block will not be chosen as a victim.

The dynamic allocator tries to write as much hot data to the SLC region as possible in order to increase the value of N_p^{hot} . The dynamic allocator also calculates a new value of α after N_p pages have been written and, for this purpose, the value of N_p^{hot} for the next time window need to be known. Similar to the approach used in our idle time prediction, we count how many hot pages were written into the SLC region during the previous 10 time windows, and use their average hotness value as N_p^{hot} for the next time window. The value of N_p^{hot} for each window can be easily measured using an update variable, which is incremented whenever a hot page is sent to the SLC region.

4.2 Improving the Endurance

To enhance the endurance of flash memory, many flash file systems adopt a special software technique called wear-leveling. In most existing wear-leveling techniques, the primary aim is to distribute erase cycles

evenly across the flash medium [11, 12]. FlexFS uses this approach, but also needs to support more specialized wear management to cope with frequent data migrations.

The use of FlexFS means that each block undergoes more erase cycles because a lot of data is temporarily written to the SLC region, waiting to move to the MLC region during idle time. To improve the endurance and prolong the lifetime, it would be better to write data to the MLC region directly, but this reduces the overall performance. Therefore, there is another important trade-off between the lifetime and performance.

To efficiently deal with this trade-off, we propose a novel wear management technique which controls the amount of data to be written into the SLC region depending on a given storage lifetime.

4.2.1 Explicit Endurance Metric

We start by introducing a new endurance metric which is designed to express the trade-off between lifetime and performance. In general, the maximum lifetime, L_{max} , of flash memory depends on the capacity and the amount of data written to them, and is expressed as follows:

$$L_{max} = \frac{C_{total} \cdot E_{cycles}}{WR}, \quad (5)$$

where C_{total} is the size of flash memory, and E_{cycles} is the number of erase cycles allowed for each block. The writing rate WR indicates the amount of data written in unit time (e.g., per day). This formulation of L_{max} is used by many flash memory manufacturers [13] because it clearly shows the lifetime of a given flash application under various environments.

Unfortunately, L_{max} is not appropriate to handle the trade-off between lifetime and performance because it expresses the expected lifetime, and not the constraints to be met in order to improve the endurance of flash memory. Instead, we use an explicit minimum lifespan, L_{min} , which represents the minimum guaranteed lifetime that would be ensured by a file system. Since FlexFS can control the writing rate WR by adjusting the amount of data written into the SLC region, this new endurance metric can be expressed as follows:

$$\begin{aligned} &\text{Control } WR \text{ by changing a wear index, } \delta \\ &\text{Subject to} \\ &L_{min} \approx \frac{C_{total} \cdot E_{cycles}}{WR}, \end{aligned} \quad (6)$$

where δ is called the wear index. In FlexFS δ is proportional to WR , and therefore δ can be used to control the value of WR . If δ is high, FlexFS writes a lot of data to the SLC region; and this increases WR due to data migrations; but if δ is low, the writing rate is reduced. Our wear management algorithm controls δ so that the lifetime specified by L_{min} is to be satisfied.

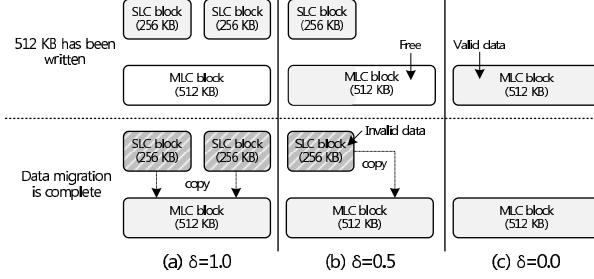


Figure 8: How the number of blocks used depends on δ

4.2.2 Assigning a Writing Budget

The proposed wear management algorithm divides the given lifetime L_{min} into n time windows $(w_0, w_1, \dots, w_{n-2}, w_{n-1})$, and the duration of each window is given as T_s . The writing rate $WR(w_i)$ for each time window w_i can also be expressed as $WB(w_i)/T_s$, where $WB(w_i)$ is the amount of data and represents the writing budget assigned to the time window w_i .

Since T_s is fixed, the assignment of a writing budget to each window significantly impacts the overall performance as well as the rate at which flash memory wears out. For example, if too large a writing budget is assigned to each window, it markedly increases the number of erase cycles for each block; on the other hand, if too small a writing budget is allocated, it lowers the overall performance. Therefore, we determine a writing budget for the window w_i as follows:

$$WB(t_i) = \frac{(C_{total} \cdot E_{cycles}) - W(t_i)}{n - (t_i/T_s)}, \quad (7)$$

where t_i is the time at the start of window w_i , and $W(t_i)$ indicates the amount of a writing budget that has actually been used by t_i . The remaining writing budget is $(C_{total} \cdot E_{cycles}) - W(t_i)$, and the number of remaining windows is $(n - (t_i/T_s))$. Therefore, the remaining writing budget is shared equally between the remaining windows. The writing budget is calculated at the beginning of every time window, so as to take changes in the workload pattern into consideration.

4.2.3 Determining the Wear Index

Once the writing budget has been assigned to a time window, the wear manager adjusts the wear index, δ , so that the amount of a writing budget actually used approximates the given writing budget. The wear index is used by a dynamic allocator, similar to Eq. (3), to distribute the incoming data across the two regions.

Figure 8 shows how the number of blocks used depends on the value of δ . The size of the SLC and MLC

blocks is 256 KB and 512 KB, respectively. Suppose that 512 KB data is written, and the data migrator moves this data from the SLC region to the MLC region. If δ is 1.0, as shown in Figure 8(a), 512 KB is written to two SLC blocks, and then the data migrator requires one MLC block to store the data from two SLC blocks. In this case, the total amount of a writing budget used is 1.5 MB because three blocks have been used for writing. If δ is 0.5, as shown in Figure 8(b), 1 MB of a writing budget is used, requiring one SLC block and one MLC block. Figure 8(c) shows the case when δ is 0.0. Only 512 KB is used because there is no data to be moved.

This simple example suggests that we can generalize the relationship between the wear index, the amount of incoming data, and the amount of a writing budget actually used, as follows:

$$IW(w_i) \cdot (2 \cdot \delta + 1) = OW(w_i), \quad (8)$$

where $IW(w_i)$ is the amount of data that arrives during the window w_i , and $OW(w_i)$ is the amount of a writing budget to be used depending on δ . In the example of Figure 8(b), $IW(t_i)$ is 512 KB and δ is 0.5, and thus $OW(t_i)$ is 1 MB. $IW(w_i) \cdot (2 \cdot \delta)$ is the amount of a writing budget used by the SLC region and $IW(w_i)$ is the amount of data to be written to the MLC region.

The wear index should be chosen so that $OW(w_i) = WB(t_i)$, and can therefore be calculated as follows:

$$\delta = \frac{WB(t_i) - IW(w_i)}{2 \cdot IW(w_i)}. \quad (9)$$

The value of δ is calculated at the beginning of w_i when the exact value of $IW(w_i)$ is unknown. $IW(w_i)$ is therefore estimated to be the average value of the previous three time windows. If $WB(t_i) < IW(w_i)$, then δ is 0, and therefore all the data will be written to the MLC region. If $IW(w_i)$ is always larger than $WB(t_i)$, it may be hard to guarantee L_{min} . However, by writing all the data to the MLC region, FlexFS can achieve a lifetime close to that of a pure MLC flash memory.

A newly determined value of δ is only used by the dynamic allocator if $\delta < \alpha$. Therefore, the wear management algorithm is only invoked when it seems that the specified lifetime will not be achieved.

4.3 Garbage Collection

The data migrator can make free blocks by moving data from the SLC region to the MLC region, but it cannot reclaim the space used by invalid pages in the MLC region. The garbage collector, in FlexFS, reclaims these invalid pages by selecting a victim block in the MLC region, and then by copying valid pages in the victim into a different MLC block. The garbage collector selects a block with many invalid pages as a victim to reduce the requirement

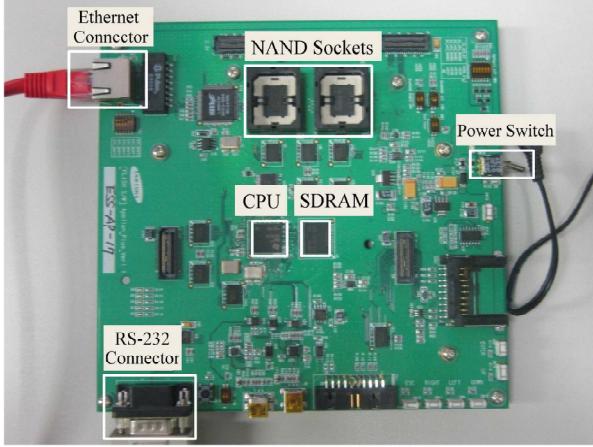


Figure 9: A snapshot of the flash development board used for experiments

for additional I/O operations, and also utilizes idle times to hide this overhead from users. Note that, it is never necessary to choose a victim in the SLC region. If cold data is stored in SLC blocks, it will be moved to the MLC region by the data migrator; but hot data need not to be moved because it will soon be invalidated.

5 Experimental Results

In order to evaluate the efficiency of the proposed techniques on a real platform, we implemented FlexFS on Linux 2.6.25.14 kernel. Our hardware system was the custom flash development board shown in Figure 9, which is based on TI's OMAP2420 processor (running at 400 MHz) with a 64 MB SDRAM. The experiments were performed on Samsung's KFXXGH6X4M-series 1-GB flash memory [4], which is connected to one of the NAND sockets shown in Figure 9. The size of each page was 4 KB and there were 128 pages in a block.

To evaluate the FlexFS file system objectively, we used two types of workload. In Section 5.1, we present experimental results from synthetic workloads. In Section 5.2, we evaluate FlexFS using actual I/O traces collected from executions of real mobile applications.

5.1 Experiments with Synthetic Workloads

5.1.1 Overall Throughput

Table 2 summarizes the configurations of the four schemes that we used for evaluating the throughput of FlexFS. In the baseline scheme, all the data is first written into SLC blocks, and then compulsorily moved to MLC blocks only when fewer than five free blocks remain. Three other schemes, BM, DA, and LA, use tech-

Table 2: Summary of the schemes used in throughput evaluation

Schemes	Baseline	BM	DA	LA
Background migration	×	○	○	○
Dynamic allocation	×	×	○	○
Locality-aware	×	×	×	○

niques to reduce the overhead of data migrations. For example, the BM scheme uses only the background migration technique, while the LA scheme uses all three proposed techniques. In all the experiments, T_{wait} was set to 1 second, N_p was 1024 pages, and T_{trig} was 15 ms. To focus on the performance implications of each scheme, the wear management scheme was disabled.

All the schemes were evaluated on three synthetic benchmark programs: *Idle*, *Busy*, and *Locality*. They were designed to characterize several important properties, such as the idleness of the system and the locality of I/O references, which give significant effects on the performance of FlexFS. The *Idle* benchmark mimics the I/O access patterns that occur when sufficient idle time is available in a system. For this purpose, the *Idle* benchmark writes about 4 MB of data (including metadata) to flash memory every 25 seconds. The *Busy* benchmark generates 4 MB of data to flash memory every 10 seconds, which only allows the I/O subsystem small idle times. The *Locality* benchmark is similar to *Busy*, except that about 25% of the data is likely to be rewritten to the same locations, so as to simulate the locality of I/O references that occurs in many applications. All the benchmarks issued write requests until about 95% of the total MLC capacity has been used. To speed up the evaluation, we limited the capacity of flash memory to 64 MB using the MTD partition manager [14].

Figure 10 compares the throughput of Baseline and BM with the *Idle* benchmark. The throughput of Baseline is significantly reduced close to 100 KB/s when the utilization approaches 50%, because before writing the

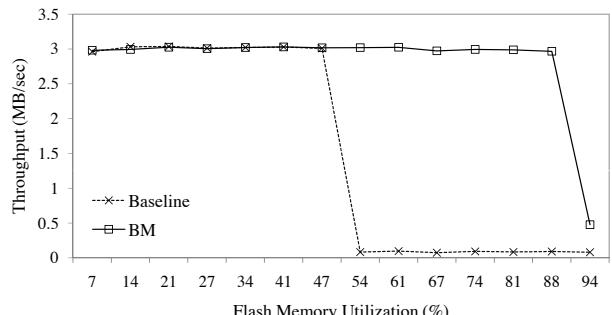


Figure 10: Performance comparison of Baseline and BM with the *Idle* benchmark

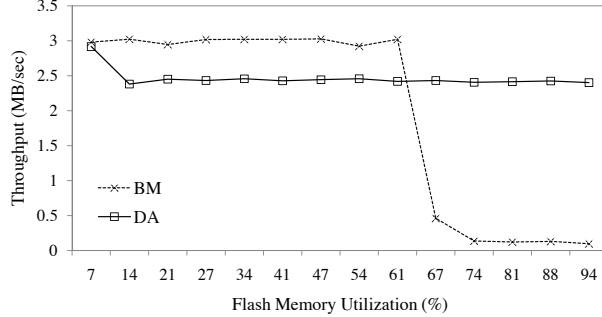


Figure 11: Performance comparison of BM and DA with the *Busy* benchmark

incoming data, the data migrator should make enough free space in the SLC region, incurring a noticeable performance degradation. However, BM achieves the same performance as SLC flash memory until the utilization exceeds 94%. Since the *Idle* benchmark allows FlexFS a lot of idle time (about 93.6% of the total execution time), it should be possible to reclaim a sufficient number of free blocks before new write requests arrive and require them. When the utilization reaches 94%, the performance of BM is significantly reduced because almost all of the available blocks is occupied by valid data, and fewer than 5 free blocks remain available.

Figure 11 compares the performance of BM and DA while running the *Busy* benchmark. In this evaluation, BM shows a better throughput than DA when the utilization is less than 67%. However, its performance quickly declines because the idle time is insufficient to allow BM to generate enough free blocks to write to the SLC region. DA does exhibit a stable write performance, regardless of the utilization of flash memory. At the beginning of the run, the value of α is initially set to 1.0 so that all the incoming data is written to the SLC region. However, since insufficient idle time is available, the dynamic allocator adjusts the value of α to 0.5. DA then writes some of the arriving data directly to the MLC region, avoiding a significant drop in performance.

Figure 12 shows the performance benefit of the locality-aware approach using the *Locality* benchmark. Note that *Locality* has the same amount of idle time compared as the *Busy* benchmark. LA achieves 7.9% more write performance than DA by exploiting the locality of I/O references. The overall write throughput of LA is 2.66 MB/s while DA gives 2.45 MB/s. The LA scheme also starts with an α value of 1.0, but that is reduced to 0.5 because the idle time is insufficient. However, after detecting a high degree of locality from I/O references, α is partially increased to 0.7 by preventing useless data migrations of hot data, and more data can then be written into the SLC region.

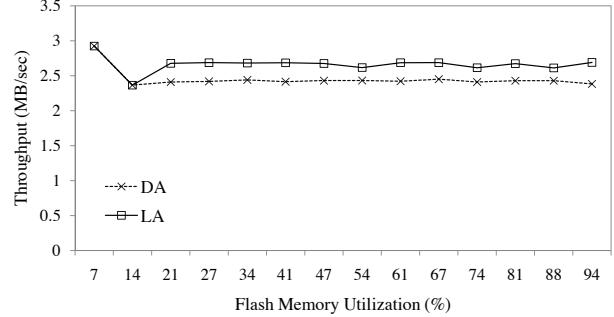


Figure 12: Performance comparison of DA and LA with the *Locality* benchmark

5.1.2 Response Time

Although the background migration contributes to improving the write throughput of FlexFS, it could incur a substantial increase in response time because I/O requests can be issued while the background migrator is running. In this subsection, to investigate the impact of the background migration on the response time, we performed evaluations with a following scenario.

We first wrote 30 MB of bulk data in order to trigger the background migrator. FlexFS was modified for all the incoming data to be written into the SLC region, regardless of the amount of idle time. After writing this data, we made 10 page write requests. The idle time between two consecutive write requests was generated using a pseudo-random number generator, but this was adjusted at least larger than T_{wait} so that all write requests were randomly issued after the background migrator has been initiated. To collect accurate and reliable results, we performed this scenario more than 30 times.

We performed our evaluation for the following four configurations. In order to know the effect of the idle time utilization, we measured the response time while varying the idle time utilization. The configurations, U_{100} , U_{50} , and U_{10} represent when FlexFS utilizes 100%, 50%, and 10% of the total idle time, respectively. This idle time utilization can be easily controlled by the value of T_{trig} . For example, the time required to move a single page from SLC to MLC is about 1.5 ms, and so the utilization of 10% can be made using T_{trig} of 15 ms. To clearly show the performance penalty from the background migration, we evaluated the response time when the background migration is disabled, which is denoted as OPT. The migration suspension mentioned in Section 4.1.1 was enabled for all the configurations.

Figure 13 shows the cumulative distribution function of the response time for the four configurations. As expected, OPT shows the best response time among all the configurations. However, about 10% of the total I/O requests requires more than 2,000 us. This response time

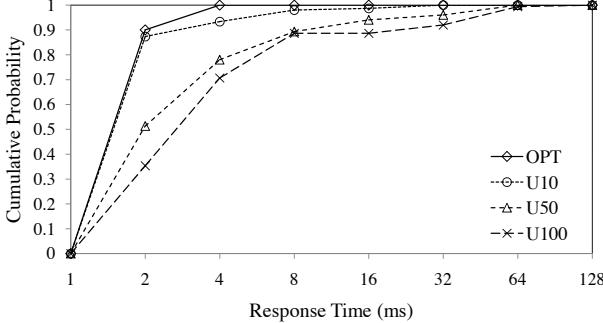


Figure 13: A comparison of response time delays on different system configurations

delay is caused by the writing of the metadata information. Although we wrote 4 KB of data into flash memory, the amount of data actually written was slightly larger than 4 KB because of the metadata overhead. Consequently, this results in additional page writes, incurring the delay in response time.

U_{10} exhibits a longer response time than OPT for about 10% of the total I/O requests, but it shows a fairly good response time. On the other hand, the performance of U_{50} and U_{100} is significantly deteriorated because they utilize a lot of idle time for data migrations, increasing the probability of I/O requests being issued while the background migrator is working. Especially, when two tasks (the foreground task and the background migration task) compete for a single CPU resource, the performance penalty caused by the resource contention is more significant than we expect.

5.1.3 Endurance

We evaluated our wear management scheme using a workload scenario in which the write patterns change over a relatively long time. We set the size of flash memory, C_{total} , to 120 MB, and the number of erase cycles allowed for each block, E_{cycles} , was 10, allowing a maximum of 1.2 GB to be written to flash memory. We set the minimum lifetime, L_{min} , to 4,000 seconds, and our wear management scheme was invoked every 400 seconds. So, there are 10 time windows, w_0, \dots, w_9 , and the duration of each, T_s , is 400 seconds. To focus our evaluation on the effect of the wear management scheme on performance, the system was given enough idle time to write all the data to the SLC region if the lifetime of flash

Table 3: The amount of data (MB) arrives for each window during the evaluation of wear management policy.

Time window	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9
Size (MB)	40	40	40	80	80	20	20	40	40	40

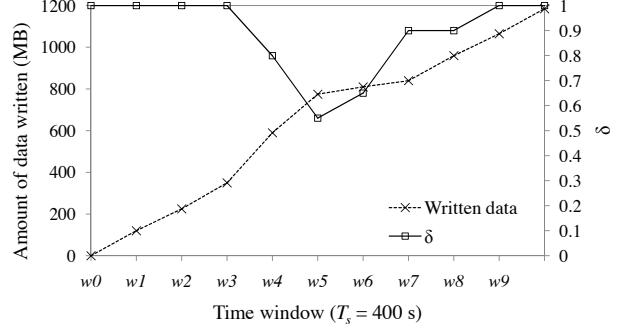


Figure 14: The changes in the size of written data and the δ value

memory is not considered.

Table 3 shows the amount of data (MB) written to flash memory for each window, w_i , and Figure 14 shows how the proposed wear management scheme adapts to changing write sizes while satisfying the minimum lifetime. Initially, FlexFS allocates a writing budget of 120 MB (= 1.2 GB / 10) to each time window. This budget is large enough to allow all the incoming data to be written to the SLC region if less than or equal to 40 MB of data arrives during each window. Therefore, during the first three windows, the value of δ is set to 1.0. During w_3 and w_4 , however, about 160 MB of data arrives, and FlexFS reduces δ to cut the migration cost. Because only 40 MB of data arrives during w_5 and w_6 , FlexFS can increase δ to give a larger writing budget to the remaining windows. We measured the amount of data written to flash memory, including extra overheads caused by migrations from the SLC region to the MLC region. FlexFS writes about 1.2 GB of data to flash memory, and thus achieving the specified minimum life span of 4,000 seconds.

We also counted the number of erase operations performed on each block while running FlexFS with and without the wear management scheme using the same workload scenario. A wear-leveling policy was disabled when the wear management scheme was not used. Figure 15 shows distributions of block erase cycles, and Table 4 summarizes the results relevant to a wear-leveling.

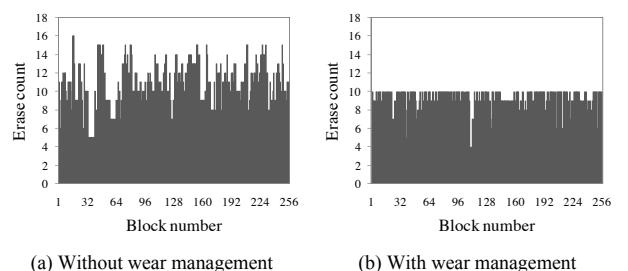


Figure 15: Distributions of block erase cycles

Table 4: Summary of results relevant to a wear-leveling

	Avg. erase cycles	Std.Dev.
w/ wear management	9.23	1.20
wo/ wear management	10.73	2.43

These results clearly indicate that with the wear management scheme FlexFS gives a good wear characteristic; the maximum erase cycle of each block is effectively limited to less than or equal to 10, and the block erase operations are evenly distributed across the flash memory medium.

5.2 Experiments with Mobile Workloads

5.2.1 Generating Mobile Workloads

In addition to the synthetic workloads discussed in Section 5.1, which were designed to evaluate one aspect of FlexFS at a time, we evaluated FlexFS using I/O traces collected from a real-world mobile platform to assess the performance of FlexFS with mobile applications.

To collect and replay I/O traces from real applications, we developed a custom mobile workload generation environment based on the Qtopia Phone Edition [15], which includes representative mobile applications such as PIMS, SMS, and media players. This environment includes three tools: a usage pattern generator, an I/O tracer, and a replayer. The usage pattern generator automatically executes mobile applications as if the user is actually interacting with applications during runtime. The I/O tracer captures I/O system calls (e.g., fopen, fread, and fwrite) while running the usage pattern generator on the Qtopia platform, and then stores collected traces in a log file. The replayer uses this log file to replay the I/O requests in our development board. Note that this log file allows us to repeat the same usage patterns for different system configurations.

For the evaluation, we executed the several mobile applications shown in Table 5 on our workload generation environment for 30 minutes. We followed a representative usage profile of mobile users reported in [16] except that more multimedia data was written in order to simulate data downloading scenario. The trace includes

Table 5: Applications used for evaluations

Application	Description
SMS	Send short messages
Address book	Register / modify / remove addresses
Memo	Write a short memo
Game	Play a puzzle game
MP3 player	Download 6 MP3 files (total 18 MB)
Camera	Take 9 pictures (total 18 MB)

Table 6: A performance comparison of FlexFS_{MLC} and FlexFS_{SLC} under mobile workloads

	Response time		Throughput
	Read (us)	Write (us)	Write (MB/s)
FlexFS _{SLC}	34	334	3.02
FlexFS _{MLC}	37	345	2.93
JFFS2	36	473	2.12

43,000 read and write requests. About 5.7 MB was read from flash memory and about 39 MB was written.

5.2.2 Evaluation Results

In order to find out whether FlexFS can achieve SLC-like performance, we evaluated the performance of two FlexFS configurations, FlexFS_{MLC} and FlexFS_{SLC}. FlexFS_{MLC} is the proposed FlexFS configuration using both SLC and MLC programming, while FlexFS_{SLC} mimics SLC flash memory by using only SLC programming. To know the performance benefits of FlexFS_{MLC}, we evaluated JFFS2 file system on the same hardware. In this subsection, we will focus on the performance aspect only, since the capacity benefit of FlexFS_{MLC} is clear.

For FlexFS_{MLC}, T_{trig} was set to 15 ms, N_p to 1024 pages, and T_{wait} to 1 second. We assumed a total capacity of 512 MB, a maximum of 10,000 erase cycles for a block, and a minimum lifetime of 3 years. The wear management policy was invoked every 10 minutes.

Table 6 compares the response time and the throughput of FlexFS_{MLC}, FlexFS_{SLC}, and JFFS2. The response time was an average over all the I/O requests in the trace file, but the throughput was measured when writing a large amount of data, such as MP3 files. Compared to JFFS2, FlexFS_{MLC} achieves 28% smaller I/O response time and 28% higher I/O throughput. However, the performance difference between FlexFS_{MLC} and JFFS2 is noticeably reduced compared to the difference shown in Table 1 because of computational overheads introduced by each file system. JFFS2 as well as FlexFS_{MLC} requires a lot of processing time for managing internal data structures, such as block lists, a metadata, and an error detecting code, which results in the reduction of the performance gap between two file systems.

The performance of FlexFS_{MLC} is very close to that of FlexFS_{SLC}. The response times of FlexFS_{MLC} are 10% and 3.2% slower for reads and writes, compared with FlexFS_{SLC}. The I/O throughput of FlexFS_{MLC} is 3.4% lower than that of FlexFS_{SLC}. This high I/O performance of FlexFS_{MLC} can be attributed to the sufficiency of idle time in the trace. Therefore, FlexFS_{MLC} can write most incoming data into the SLC region, improving the overall I/O performance.

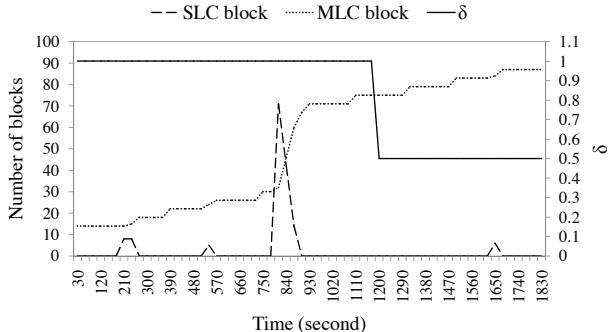


Figure 16: The changes in the number of SLC and MLC blocks with a mobile workload in FlexFS_{MLC}

The graph in Figure 16 shows in more detail how FlexFS_{MLC} achieves I/O efficiency. We counted the number of each type of block every 30 seconds. In the graph, the regions around 840 seconds clearly demonstrate the effectiveness of the proposed techniques. Starting from 750 seconds, many MP3 files of about 18 MB are intensively written into flash memory. FlexFS_{MLC} can write all this data into the SLC region because the idle time predictor in the dynamic allocator predicts there will be enough idle time, which allows aggressive writes to the SLC region.

From our observations on the representative mobile workloads, there are two distinctive characteristics in I/O access patterns. First, many mobile embedded systems such as mobile phones and smart phones are likely to have sufficient idle time; the average idle time accounts for about 89% of the total execution time. Second, most data is intensively written to flash memory within a short time interval. As the experimental results show, FlexFS is effectively designed for dealing with such characteristics, and thus can achieve the I/O performance close to SLC flash memory.

The small performance penalty of FlexFS_{MLC} results from ensuring the given minimum lifetime. As shown in Figure 16, at around 1,200 seconds the wear management policy reduces the value of δ to 0.5, which degrades the write performance of FlexFS_{MLC}. However, this decision was necessary because a large number of writes to the SLC region for storing several MP3 files reduced the number of erase cycles significantly. To meet the required minimum lifetime, FlexFS wrote 50% of the data to the MLC region directly. This result indicates that the poor wear characteristic of MLC flash memory could be a hurdle for FlexFS to achieve its performance benefit.

However, it must be noted that 512 MB of flash capacity used in our evaluation is very small compared to commercial flash applications. Actually, many flash devices already employ several GB of flash memory and its capacity doubles every two or three years. For exam-

ple, if a flash device has 16 GB MLC flash memory and the minimum lifetime is set to 3 years, the writing budget per day is about 146 GB. Therefore, it may safely be assumed that the endurance problem would be mitigated without a significant performance degradation.

6 Related Work

Many file systems for NAND flash memory have been studied in recent years. JFFS2 [5] and YAFFS [7] are representative, and are both the log-structured file systems [6], which write data sequentially to NAND flash memory. JFFS2 was originally developed for NOR flash memory, and later extended to NAND devices. JFFS2 stores metadata and regular data together. YAFFS is similar to JFFS2 except that metadata is stored in a spare area of each page to promote fast mounting of the file system. They are both designed for the homogeneous flash memory media, and do not support the heterogeneous flash memory devices discussed in this paper.

Recently, there have been several efforts to combine both SLC and MLC flash memory. Chang et al. suggest a solid-state disk which is composed of a single SLC chip and many MLC chips [17], while Park et al. present a flash translation layer for mixed SLC-MLC storage systems [18]. The basic idea of these two approaches is to store frequently updated data in the small SLC flash memory while using the large MLC flash memory for storing bulk data. This brings the overall response time close to that of SLC flash memory while keeping the cost per bit as low as MLC flash memory. However, these approaches cannot break down when a large amount of data has to be written quickly, because they only use the small SLC flash memory so as to achieve their cost benefit. In this situation, the overall I/O throughput will be limited to the throughput of MLC flash memory. But FlexFS can handle this case efficiently by flexibly increasing the size of the SLC region, and therefore combines the high performance of SLC flash memory with the high capacity of MLC flash memory.

The hybrid hard disk [19, 20] is another heterogeneous storage system which uses flash memory as a non-volatile cache for a hard disk. In a hybrid hard disk, flash memory is used to increase the system responsiveness, and to extend battery lifetime. However, this approach is different from our study in which it does not give any considerations on optimizing the storage system by dynamically changing its organization.

7 Conclusions

FlexFS is a file system that takes advantage of flexible programming of MLC NAND flash memory. FlexFS is

designed to maximize I/O performance while making the maximum capacity of MLC flash memory available. The novel feature of FlexFS is migration overhead reduction techniques which hide the incurred migration overhead from users. FlexFS also includes a novel wear management technique which mitigates the effect of the data migration on the lifetime of flash memory. Experimental results show that FlexFS achieves 90% and 96% of the read and write performance of SLC flash memory with real-world mobile workloads.

There are a few areas where FlexFS can be further improved. First, even though the background migration is effective in hiding the migration overhead, it is less efficient from the energy consumption perspective because it reduces the probability that the system enters a low-power state. In order to better handle both the performance and energy consumption simultaneously, we are developing a dynamic allocation policy that takes into account an energy budget of a system. Second, for FlexFS to be useful on a wide range of systems, the poor wear characteristic of MLC flash memory should be addressed properly. To handle this problem, we are also investigating a wear management policy for a storage architecture in which SLC flash memory is used as a write buffer for MLC flash memory.

8 Acknowledgements

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (No. R0A-2007-000-20116-0) and the Brain Korea 21 Project in 2009. This work was also supported by World Class University (WCU) program through KOSEF funded by the Ministry of Education, Science and Technology (No. R33-2008-000-10095-0). Samsung Electronics partially supported our FlexFS research and the ICT at Seoul National University provided research facilities for this study.

References

- [1] F. Roohparvar, "Single Level Cell Programming in a Multiple Level Cell Non-volatile Memory Device," In *United States Patent, No 11/298,013*, 2007.
- [2] M. Bauer, "A Multilevel-Cell 32 Mb Flash Memory," In *Proceedings of the Solid-State Circuits Conference*, February 1995.
- [3] P. Pavan, R. Bez, P. Olivo, and E. Zanoni, "Flash Memory Cells - An Overview," In *Proceedings of the IEEE*, vol. 85, no. 8, 1997.
- [4] Samsung Electronics Corp., "Flex-OneNAND® Specification," http://www.samsung.com/global/system/business/semiconductor/product/2008/2/25/867322ds_kfxxgh6x4m_rev10.pdf.
- [5] D. Woodhouse, "JFFS : The Journalling Flash File System," In *Proceedings of the Linux Symposium*, July 2001.
- [6] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, no. 1, 1992.
- [7] Aleph One, "YAFFS: Yet Another Flash File System," <http://www.yaffs.net/>, 2002.
- [8] L. Benini, A. Bogliolo, and G. D. Micheli, "A Survey of Design Techniques for System-level Dynamic Power Management," *IEEE Transactions on VLSI Systems*, vol. 8, no. 3, 2000.
- [9] E. Chan, K. Govil, and H. Wasserman, "Comparing Algorithms for Dynamic Speed-setting of a Low-power CPU," In *Proceedings of the Conference on Mobile Computing and Networking (MOBICOM '95)*, November 1995.
- [10] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," In *Proceedings of the Conference on Management of Data (SIGMOD '93)*, May 1993.
- [11] H. Kim and S. Lee, "An Effective Flash Memory Manager for Reliable Flash Memory Space Management," *IEICE Transactions on Information and System*, vol. E85-D, no. 6, 2002.
- [12] L. Chang and T. Kuo, "Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation," *ACM Transactions on Storage*, vol. 1, no. 4, 2005.
- [13] SanDisk, "Longterm Data Endurance (LDE) for Client SSD," http://www.sandisk.com/Assets/File/pdf/oem/LDE_White_Paper.pdf, 2008.
- [14] Memory Technology Device (MTD), <http://www.linux-mtd.infradead.org/doc/general.html>.
- [15] Nokia Corp., "Qtopia Phone Edition 4.1.2," <http://www.qtsoftware.com/products/>.
- [16] H. Verkasalo and H. Hämmänen, "Handset-Based Monitoring of Mobile Subscribers," In *Proceedings of the Helsinki Mobility Roundtable*, June 2006.
- [17] L.P. Chang, "Hybrid Solid-State Disks: Combining Heterogeneous NAND Flash in Large SSDs," In *Proceedings of the Conference on Asia and South Pacific Design Automation (ASP-DAC '08)*, January 2008.
- [18] S. Park, J. Park, J. Jeong, J. Kim, and S. Kim, "A Mixed Flash Translation Layer Structure for SLC-MLC Combined Flash Memory System," In *Proceedings of the Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED '08)*, February 2008.
- [19] R. Panabaker, "Hybrid Hard Disk and ReadyDrive Technology: Improving Performance and Power for Windows Vista Mobile PCs," In *Proceedings of the Microsoft WinHEC*, May 2006.
- [20] Y. Kim, S. Lee, K. Zhang, and J. Kim, "I/O Performance Optimization Technique for Hybrid Hard Disk-based Mobile Consumer Devices," *IEEE Transactions on Consumer Electronics*, vol. 53, no. 4, 2007.

Layering in Provenance Systems

Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland,
Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, Robin Smogor
Harvard School of Engineering and Applied Sciences
pass@eecs.harvard.edu

Abstract

Digital provenance describes the ancestry or history of a digital object. Most existing provenance systems, however, operate at only one level of abstraction: the system call layer, a workflow specification, or the high-level constructs of a particular application. The provenance collectable in each of these layers is different, and all of it can be important. Single-layer systems fail to account for the different levels of abstraction at which users need to reason about their data and processes. These systems cannot integrate data provenance across layers and cannot answer questions that require an integrated view of the provenance.

We have designed a provenance collection structure facilitating the integration of provenance across multiple levels of abstraction, including a workflow engine, a web browser, and an initial runtime Python provenance tracking wrapper. We layer these components atop provenance-aware network storage (NFS) that builds upon a Provenance-Aware Storage System (PASS). We discuss the challenges of building systems that integrate provenance across multiple layers of abstraction, present how we augmented systems in each layer to integrate provenance, and present use cases that demonstrate how provenance spanning multiple layers provides functionality not available in existing systems. Our evaluation shows that the overheads imposed by layering provenance systems are reasonable.

1 Introduction

In digital systems, *provenance* is the record of the creation and modification of an object. Provenance provides answers to questions such as: *How does the ancestry of two objects differ? Are there source code files tainted by proprietary software? How was this object created?* Most existing provenance systems operate at a single level of abstraction at which they identify and

record provenance. Application-level systems, such as Trio [29], record provenance at the semantic level of the application – tuples for a database system. Other application-level solutions record provenance at the level of business objects, lines of source code, or other units with semantic meaning to the application. Service-oriented workflow (SOA) approaches [8, 9, 23], typically associated with workflow engines, record provenance at the level of workflow stages and data or message exchanges. System-call-based systems such as ES3 [3], TREC [28], and PASS [21] operate at the level communicated via system calls – processes and files. In all of these cases, provenance increases the value of the data it describes.

While the provenance collected at each level of abstraction is useful in its own right, integration across these layers is crucial but currently absent. Without a unified provenance infrastructure, individual components produce islands of provenance with no way to relate provenance from one layer to another. The most valuable provenance is that which is collected at the layer that provides user-meaningful names. If users reason in terms of file names, then a system such as PASS that operates at the file system level is appropriate. If users want to reason about abstract datasets manipulated by a workflow, then a workflow engine’s provenance is appropriate. As layers interoperate, the layers that name objects produce provenance, transmitting it to other layers and forming relationships with the objects at those different layers. For example, this might associate many files that comprise a data set with the single object representing the data set. PASS captures provenance transparently without application modification, but it might not capture an object’s semantics. If applications encapsulate that semantic knowledge, then those applications require modification to transmit that knowledge to PASS. In summary, application and system provenance provide different benefits; the value of the union of this provenance is greater than the sum of its parts.

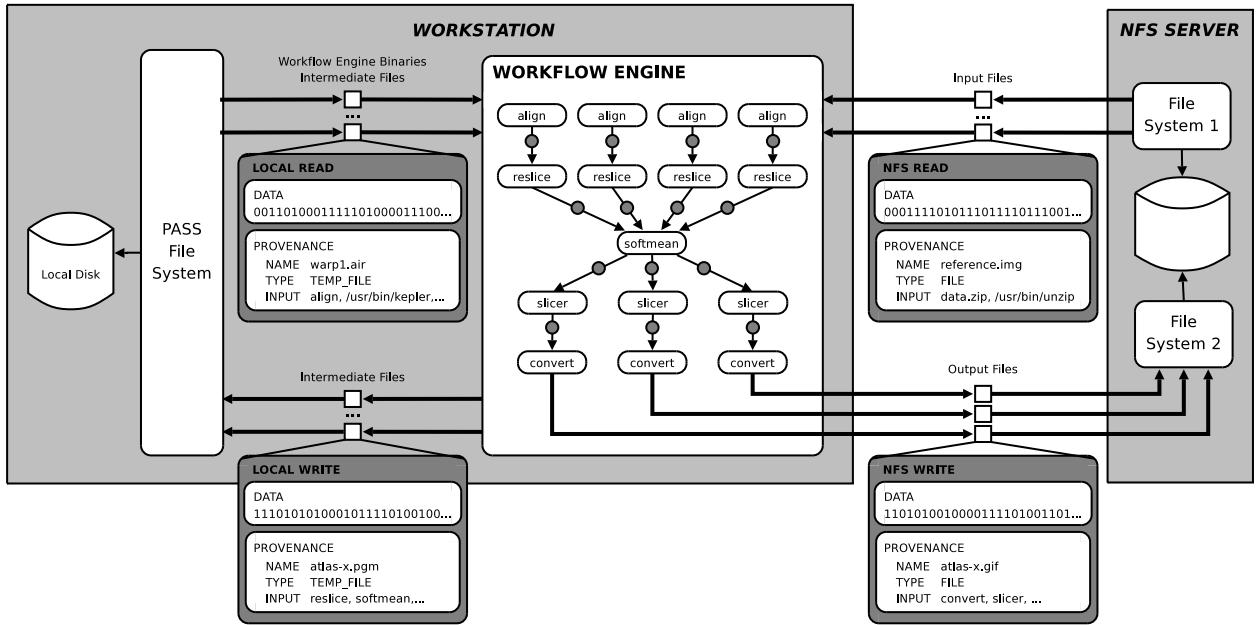


Figure 1: **Example of a layered architecture introduced in Section 1.** This scenario demonstrates a workflow application running on a workstation, but accessing its inputs and outputs on remote file servers. The workflow engine, local file system, and remote file system all capture provenance, but only by integrating that provenance can we respond to queries that require complete ancestry of an object.

Consider the scenario in Figure 1. A workflow engine running on a workstation reads input files from an NFS-mounted file system, runs a workflow that produces intermediate files stored on the the workstation’s disk, and ultimately produces three output files that are stored on a second NFS-mounted file system. Imagine that we run the workflow on Monday, and unbeknownst to us, a colleague modifies one of the input files on Tuesday. When we run the workflow on Wednesday, we find that it produces different results. If we capture provenance only on the local or remote file systems, we cannot see any of the processing stages that occur inside the workflow engine. If we capture provenance inside the workflow engine, then we lose track of the fact that one of the inputs to the workflow changed, since it changed in a manner transparent to the workflow engine (on a remote server). Properly tracing the ancestry of the output file requires the full provenance chain: from one remote server through the local system and workflow engine to another remote server.

This example illustrates the challenges that we encountered in developing a layered provenance architecture. We will use this example throughout the rest of the paper to discuss these challenges and describe our solutions to them. We describe the PASSv2 system, a new version of PASS that enables the seamless integration of provenance from different layers of abstraction. In

such a layered system, data and provenance flow together through the different layers. The resulting system provides a unified provenance infrastructure allowing users to get answers to queries that span layers of abstraction, including information about objects in different applications, the local file system, remote file systems, and even those downloaded from the Web. Furthermore, the system allows an arbitrary number of layers to stack on each other (we demonstrate a three layer stack in Section 7).

The contributions of this work are:

- Use cases demonstrating the utility of integrating provenance-aware applications and a provenance-aware operating system.
- An architecture for provenance-aware systems that integrates provenance across multiple abstraction layers.
- A prototype demonstrating the capabilities of this layered architecture through the creation of an integrated set of provenance-aware components: NFS, a workflow engine, a browser (`links`), and an initial runtime Python provenance tracking wrapper.

The rest of this paper is organized as follows: In Section 2, we place this work in the context of existing systems and introduce the idea of integrating provenance across semantic layers in a system. In Section 3, we present use cases that highlight the benefits

of layering. In Section 4, we discuss the fundamental challenges inherent in building systems that integrate provenance across multiple layers, and in Section 5, we present our layered architecture, demonstrating how it addresses these challenges. In Section 6, we describe the provenance-aware applications and remote file server we developed. Section 7 presents the cost of providing these features in terms of time and space overheads. In Section 8, we discuss related work, and we conclude in Section 9.

2 Background

Previous work exists at each of the layers discussed in the introduction, but there is no approach that integrates across the different levels of abstraction to provide a unified solution.

At the domain-specific level, systems like GenePattern [10] provide provenance for environments in which biologists perform routine analyses. Experiments done within the analysis environment record and maintain information about the particular algorithms and parameters used, but this information can be lost if data is manipulated outside of the environment.

Tracking provenance at the level provided by workflow engines – such as Pasoa [23], Chimera [9], and Kepler [2] – allows users to group collection of related objects into single logical entities. For example, scientists frequently refer to logical data sets containing hundreds or thousands of individual files. These systems can answer queries such as: *What were all the output files of a particular experiment?* or *What version of the software release are we using for this analysis?* These systems lose some of the semantic knowledge available at the domain-specific level, but do provide the ability to handle abstractions such as data sets.

System level solutions like ES3 [3] and PASSv1 [21] capture information at the operating system level, losing both the semantic information of domain-specific solutions and also the relationships among data sets and processing units found in workflow engines. However, these systems provide a wealth of information about the environment in which objects are created, such as the specific binaries, libraries, and kernel modules in use.

Provenance is not the same as a security label, and provenance systems are different from label-based security systems such as HiStar [31], Asbestos [6], and Flume [18]. These systems track information flow but not in sufficient detail for general provenance querying. ES3 and PASSv1 capture not only the *fact* of relationships among data sets but also the *means*, including data such as process arguments and environment variables, and support queries over this information.

All of these solutions fundamentally fail to account for the different levels of abstraction at which users need to reason about their data and processes. Users should be able to work at any or all levels as desired, rather than being limited to one. This requires being able to relate objects that appear in one layer to their manifestations in other layers.

It seems that perhaps it is sufficient for each system to generate its own provenance independently and then use the names of objects to link the layers together, as by a relational join. However, the Second Provenance Challenge [25] showed that even at a single level of abstraction, uniform object naming is both fundamental to provenance interoperability and nontrivial. Across abstraction boundaries, it is harder yet. For example, an object that exists in one layer may in other layers have some local manifestation, such as a collection of files forming a data set, but no clearly defined name. Thus, using only object names or identifiers is not sufficient.

Providing a uniform basis for object identity and linkage requires an integrated provenance solution with explicit layer support, where data moving from one layer to the next carries its provenance with it.

3 Use Cases

We have explored provenance collection in a variety of different contexts ranging from NFS to web browsers. We began with NFS, as a large number of users store their data on network-attached storage. Developing provenance-aware NFS (PA-NFS) helped us understand how to extend provenance outside a single machine. Next, we decided to explore integrating a provenance-aware workflow engine with PASSv2 as most prior provenance systems operated at the level of a workflow engine [2, 9, 23]. We selected the Kepler open-source workflow enactment engine [2] in which to explore integrating workflow provenance with PASSv2. We then explored adding provenance collection to an application that bears little similarity to workflow engines and operating systems: a web browser. We used the *links* text based web browser for this purpose. Last, we built a set of Python wrappers to capture provenance for Python applications.

In this section, we present scenarios and differentiate the problems that provenance-aware systems can address with and without layering.

3.1 Provenance Aware NFS

Use Case: Finding the Source of Anomalies

Scenario: Implementing the scenario depicted in Figure 1, we use Kepler to execute the Provenance Challenge

workflow [24], reading inputs from one NFS file server and writing outputs to another. Between two executions, unbeknownst to us someone modifies an input file. When we examine the new output, we see that it is different from the old output, and we would like to understand why.

Without Layering: If we examine only the Kepler provenance, we would think that the two executions were identical, since the change in the input file is invisible to Kepler. If we examine only the PASSv2 provenance, we would see that there was a different input to the Kepler workflow, but we would not know for sure that the input was actually used to produce the output since we cannot see how the multiple inputs and outputs are related.

With Layering: If Kepler runs on PASSv2, then the PASSv2 provenance store contains the provenance from both systems. Hence, it is possible to both determine and verify that the input file modification was responsible for the different output.

3.2 Provenance-Aware links

Use Case: Attribution

Scenario: A professor is preparing a presentation and has a number of graphs and quotes that have been previously downloaded from the Web. She copies these objects into the directory containing the presentation. Now, she would like to include proper attribution for them, but none remain in the browser’s history and some of them are no longer even accessible on the Web.

Without Layering: Any browser can record the URL and name of a downloaded file and, when the site is revisited, can verify if the file has changed. (In fact, this is how most browser caches function.) However, if the user moves, renames, or copies the file, the browser loses the connection between the file and its provenance. The provenance collected by PASSv2 alone is insufficient as it only records the fact that the file was downloaded by the browser.

With Layering: A provenance-aware browser generates provenance records that include the URL of the downloaded file and transmits them to PASSv2 when it writes the file to disk. PASSv2 ensures that the file and its provenance stay connected even if it was renamed or copied. Our absent-minded professor can now determine the browser provenance for the file included in the presentation.

Use Case: Determining Malware Source

Scenario: Suppose Alice downloaded a codec from a web site. Suppose further that Eve, unbeknownst to Alice, has hacked the web site and caused this codec to contain malware. Alice later discovers that her computer

has been infected. She would like to be able to find the origin of the malware and the extent of the damage.

Without Layering: Alice can traverse the provenance graph recorded by PASSv2 (similar to Backtracker [17] and Taser [11]) to identify and remove the malware binaries and recover any corrupted files. However, PASSv2 by itself cannot identify the web site from which the malware was downloaded. Conversely, a provenance-aware browser can identify the web site from which a known malware file was downloaded, but it cannot track the spread of that malware through the file system.

With Layering: A provenance-aware browser integrated with PASSv2 can help identify the web site that Alice was visiting when malware was downloaded, any linked third party site where the malware download originated, as well as other details about the browsing session (for example, the user may have been redirected from a trusted site). It can also find other files and descendants of files downloaded from the same web site, which may now be suspect. Layering with PASSv2 also provides an extra level of protection. The malware can compromise the browser, but to hide the fact that it was ever on the system, it also needs to compromise the operating system. If the operating system is compromised, we can ensure the integrity of the provenance collected before the compromise by using a selective versioning secure disk system [27].

3.3 Provenance-Aware Python

Use Case: Determining Data Origin

Scenario: Through approximately 400 experiments on 60 specimens over the course of a week, colleagues in Iowa State’s Thermography Research Group developed a set of data quantitatively relating crack heating to the vibrational stresses on the crack. The experiment logs for these data were stored in a series of XML files by the team’s data acquisition system. A team member developed a Python script to plot crack heating as a function of crack length for two different classifications of vibrational stress. Our goal was to determine the sources of the specific XML data files that contributed to each plot.

Without Layering: This might have been a simple problem for PASSv2, except that the analysis program reads in *all* the XML data files to determine which ones to use. PASSv2 reports that the plot derives from all the XML files. Provenance-Aware Python knows which XML documents were actually used, but it does not know the source files of those documents.

With Layering: In a layered Provenance-Aware Python/PASSv2 system, queries over the provenance of the resulting plot can report both the precise XML documents, the files from which they came, and the prove-

nance of those files.

Use Case: Process Validation

Scenario: They upgraded the Python libraries on one of their analysis machines, introducing bugs in a calculation routine used to estimate crack heating temperatures. The group discovered this bug after running the experiments and wanted to identify the results that were affected by the erroneous routine.

Without Layering: PASSv2 can distinguish which output files were generated using the new Python library, but cannot determine which of those files were generated by invoking the erroneous routine. Provenance-Aware Python can determine which files were generated by invoking the calculation routine, but cannot tell which version of Python library was used.

With Layering: Integrating the provenance collected by PA-Python and PASS identifies the files that have incorrect data, because they descend from both the new Python library and the calculation routine.

4 Challenges in Layering

Mapping objects between the different layers of abstraction is only one of the challenges facing layered provenance systems. We identified six fundamental challenges for a layered provenance architecture:

Interfacing Between Provenance-Aware Systems: The manner in which different provenance-aware systems stack is not fixed. A workflow engine might invoke a provenance-aware Python program (see Section 6.4) in one instance and in another instance be invoked by it. Thus, provenance-aware components must be able to both accept and generate messages that transmit provenance. We designed a single universal API appropriate for communication among PASSv2 components and also among different provenance systems. It took several iterations to develop an API that was both general and simple; we discuss the resulting Disclosed Provenance API (DPAPI) in Section 5.2.

Object Identity: As mentioned earlier, objects may be tangible at one layer and invisible at another. Imagine tracking provenance in a browser, as in Section 6.3. It would be useful to track each browser session as an independent entity. However, browser sessions do not exist as objects in the file system, so it is not obvious how to express a dependence between a browser page and a file downloaded from it. We show how the DPAPI makes objects from one layer visible to other layers and how the *distributor* lets us manage objects that are not manifest at a particular layer in Section 5.5.

Consistency: Provenance is a form of metadata; we need to define and enforce consistency semantics be-

tween the data and its metadata, so users can make appropriate use of it. The DPAPI bundles data and provenance together to achieve this consistency and our layered file system, Lasagna (Section 5.6), maintains this consistency on disk.

Cycles: In earlier work, we discussed the challenge of detecting and removing cycles in PASSv1 [21]. This problem becomes even more complicated in a layered environment. Since there are objects that appear at one layer and not at others, we may need to create relationships between objects that exist in different layers, and then detect and remove cycles that these relationships introduce. In Section 5.4, we discuss how the *analyzer* performs cross-layer cycle detection.

Query: Collecting provenance is not particularly valuable if we cannot make it available to a user or administrator in a useful fashion. We shadowed several computational science users to understand what types of queries they might ask a provenance system. After struggling through three generations of query languages for provenance, we incorporated the input from our users and derived the following list of requirements for a provenance query language [16]:

- The basic model should be *paths through graphs*;
- Paths should be first-class language level objects;
- Path matching should be by regular expressions over graph edges; and
- The language needs sub-queries and aggregation.

Query languages for semi-structured data proved the best match; our query language PQL (Path Query Language) derives from one of these. Section 5.7 provides a brief overview.

Security: While there has been research showing the use of provenance for auditing and enhancing security [13], there has been little work on security controls for the provenance itself. The fundamental provenance security problem is that provenance and the data it describes do not necessarily share the same access control. There is no universally correct rule that dictates which of the two (data or provenance) requires stronger control. For example, consider a report generated by aggregating the health information of patients suffering a certain ailment. While the report (the data) can be accessible to the public, the files that were used to generate the report (the provenance) must not be. The provenance must be more tightly controlled than the data. Conversely, a document produced by a government panel (the data) might be classified, but the membership of the committee and the identities of all participants in briefings (the provenance) may nonetheless be entirely public. The data carries stronger access control than the provenance. Creating an access control model for provenance is outside the scope of this

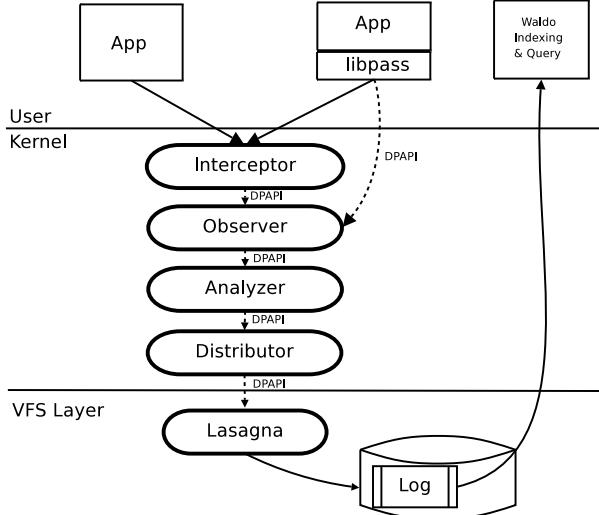


Figure 2: PASSv2 Architecture

paper; however, a related paper presents an in-depth discussion of the problem and our approach to solving it [4].

5 Architecture

We begin with a high level overview that introduces the main components of the PASSv2 system. Then we explain each component of the system in detail and show how it addresses the challenges discussed above.

5.1 Overview

From a user perspective, PASSv2 is an operating system that collects provenance invisibly. Application developers can also use it to develop provenance-aware applications. Figure 2 shows the seven main components of the PASSv2 system. These are:

libpass: libpass is a library that exports the DPAPI to user-level. Application developers develop provenance-aware applications by augmenting their code to collect provenance and then issuing DPAPI calls to libpass.

Interceptor: The interceptor intercepts system calls and passes information to the observer.

Observer: The observer translates system call events to *provenance records*. For example, when a process P reads a file A , the observer generates a record $P \rightarrow A$, indicating that P depends on A . Hence, together the observer and the interceptor generate provenance.

Analyzer: The analyzer processes the stream of provenance records and eliminates duplicates and ensures that cyclic dependencies do not arise.

Distributor: The distributor caches provenance for objects that are not persistent from the kernel’s perspective, such as pipes, processes and application-specific objects (e.g., a browser session or data set) until they need to be materialized on disk.

Lasagna: Lasagna is the provenance-aware file system that stores provenance records along with the data. Internally, it writes the provenance to a log. The log format ensures consistency between the provenance and data appearing on disk.

Waldo: Waldo is a user-level daemon that reads provenance records from the log and stores them in a database. Waldo is also responsible for accessing the database on behalf of the query engine.

5.2 Disclosed Provenance API (DPAPI)

The DPAPI is the central API inside PASSv2. It allows transfer of provenance both among the components of the system and between layers. Applications use the DPAPI to send (“disclose”) provenance to the kernel. The same interface is used to send provenance to the file system. The DPAPI consists of six calls: `pass_read`, `pass_write`, `pass_freeze`, `pass_mkobj`, `pass_reviveobj`, and `pass_sync` and two additional concepts: the *pnode number* and the *provenance record*.

A pnode number is a unique ID assigned to an object at creation time. It is a handle for the object’s provenance, akin to an inode number, but never recycled. A provenance record is a structure containing a single unit of provenance: an attribute/value pair, where the attribute is an identifier and the value might be a plain value (integer, string, etc.) or a cross-reference to another object. Provenance records may contain *ancestry* information, records of data flows, or *identity* information.

The `pass_read` and `pass_write` operations are like `read` and `write` but are provenance-aware. This ensures that provenance and data move together, providing consistency of provenance and data as required by Section 4.

The `pass_read` call returns both the data requested and the exact identity of what was read: the file’s *pnode* number and version as of the moment of the read. This ensures that applications or other higher layers can construct provenance records that accurately describe what they read, also a critical component of consistency.

The `pass_write` call takes both a data buffer and a “bundle” of provenance records that describe the data. A provenance bundle is an array of object handles and records, each potentially describing a different object. The complete provenance for a block of data written to a file might involve many objects (e.g., several processes

and pipes in a shell pipeline). This organization allows all of the separate objects to be sent as a single unit.

Cycle-breaking sometimes requires creating new versions of objects. In a layered system, versions must be handled at the bottom level (the storage system), but cycle-breaking may occur at any level. The `pass_freeze` call breaks cycles by requesting a new version.

As discussed in Section 4, provenance-aware applications may need to represent objects, such as browser sessions, data sets, or program variables, that do not map to a particular file system object. The `pass_mkobj` call allows applications to create such objects. These objects are referenced like files, with file handles. The objects can also be used to relate names/objects at one level to names/objects at another level. A system at any layer can create objects using `pass_mkobj` and create dependencies between its objects and objects at different layers of abstraction by issuing `pass_write` calls. Users can then issue queries using the name in the most convenient abstraction layer (e.g., filename) and PASSv2 can retrieve the appropriate objects across the layers using these dependencies.

We initially designed the objects returned by `pass_mkobj` to be transient and applications had no means to access these objects again after closing them. However, when developing provenance-aware applications, we discovered occasions where we needed to access these objects. Hence, we added the `pass_reviveobj` call that takes a pnode number and version and returns an object previously created via `pass_mkobj`.

By default, the provenance associated with an object returned via `pass_mkobj` is not flushed to disk unless it becomes a part of the ancestry of a persistent object on a PASS-enabled volume. This is correct behavior for purely transient objects with no descendants (e.g., processes), but it would lose objects that exist only at layers above PASS. Applications can use the `pass_sync` function to make persistent provenance associated with an object created via `pass_mkobj` even if it is not in the ancestry of a persistent PASS-volume object.

Applications link against `libpass` to use the user-level DPAPI to record provenance. Such applications are *provenance-aware*. The DPAPI enables an arbitrary number of layers of provenance-aware applications. For example, we can construct a system with five layers using a provenance-aware Python application that uses a provenance-aware Python library, both of which execute on a provenance-aware Python interpreter. That provenance-aware Python interpreter might then use a PA-NFS utilizing PASSv2. Note that the provenance-aware library and provenance-aware interpreter both accept DPAPI calls from *higher* layers and issue DPAPI

calls to *lower* layers.

5.3 Provenance Generation

Provenance generation involves two system components: the *interceptor* and *observer*. The interceptor captures system call events and reports them to the observer. The PASSv2 interceptor handles the following system calls: `execve`, `fork`, `exit`, `read`, `readv`, `write`, `writev`, `mmap`, `open`, and `pipe`, and the kernel operation `drop_inode`. The interceptor is a thin operating system specific layer, while the remaining system components can be mostly operating system independent.

The observer takes the information it receives from the interceptor, constructs provenance records, and passes those records to the analyzer via DPAPI calls. For example, when a process issues a `read` system call, the observer first issues a `pass_read` on the file. When the `pass_read` returns with the data, pnode, and version of the file, the observer creates a record stating that the particular version of the file is an input to the process, thereby creating a dependency between the process and the file. It then sends the record to the analyzer by issuing a `pass_write` with the provenance record, but no data. When that process then issues a `write` system call, the observer creates a record stating that the process is an input to the written file and issues a `pass_write` containing both this provenance record and the data from the `write` system call, thereby creating a dependency between the process and the file.

The observer is also the entry point for provenance-aware applications that use the DPAPI to explicitly disclose provenance records to PASSv2. The observer is the appropriate entry point, since PASS might need to generate additional provenance records even when an application is disclosing provenance. For example, when an application invokes a `pass_write` DPAPI call, apart from the explicit provenance disclosed by the application, the observer has to create a record that captures the dependency between the application and the file. The observer converts the provenance that higher-level provenance-aware applications explicitly disclose via DPAPI calls into appropriate kernel structures and passes the records to the analyzer.

5.4 Analyzer

The *analyzer* eliminates redundant provenance and cycles in the stream of provenance records that it receives. Programs generally perform I/O in relatively small blocks (e.g., 4 KB), issuing multiple reads and writes when manipulating large files. Each `read` or `write` call causes the observer to emit a new record, most of which are identical. The analyzer removes such

duplicates. Meanwhile, cycles can occur when multiple processes are concurrently reading and writing the same files. The analyzer prevents cycles by creating new versions of objects. In PASSv1, we used an algorithm that maintains a global graph of object dependencies and explicitly checks for cycles. On detecting a cycle, the algorithm merged all the nodes in the cycle into a single entity. This proved challenging, and there were cases where we were not able to do this correctly. In PASSv2, we use a more conservative algorithm, called the *cycle avoidance* algorithm that uses only an object’s local dependency information to avoid cycles. We discuss and analyze this algorithm in detail in earlier work [20]. Since any semantic information that the higher-level applications disclose to PASSv2 is via objects returned through `pass_mkobj`, the analyzer works in a layered environment without modification.

5.5 Distributor

Since processes are first-class objects, the system must track and store their provenance. However, processes are not by themselves persistent objects residing on a PASS-enabled volume. Where should their provenance be stored? Similar issues arise with pipes, files from non-PASS volumes, and objects introduced by provenance-aware applications. In these cases, PASSv2 must select some PASS-enabled volume on which to store their provenance. The *distributor* addresses this issue.

The distributor caches provenance records for all objects that are not PASS files. When those objects become part of the ancestry of a persistent object on a PASS-enabled volume or are explicitly flushed via `pass_sync`, the distributor assigns these objects to a PASS volume (either that of the persistent ancestor or the one specified when an object was created) and flushes the provenance records by issuing a `pass_write` to Lasagna.

5.6 Lasagna & Waldo

Lasagna is our provenance-aware file system that stores both provenance and data. Lasagna is a stackable file system, based upon the eCryptfs [12] code base. Lasagna implements the DPAPI interface in addition to the regular VFS calls. We implement `pass_read`, `pass_write`, `pass_freeze` as inode operations and `pass_mkobj` and `pass_reviveobj` as superblock operations.

PASSv1 wrote provenance directly into databases that provided indexed access to provenance. This arrangement was neither flexible nor scalable, so PASSv2 writes all provenance records to a log. A user-level daemon process, *Waldo*, later moves the provenance to a database and indexes it. When the log file exceeds a parametrized

maximum size or has been dormant for a parametrized length of time, the kernel closes the log and creates a new one. Waldo uses the Linux `inotify` interface to monitor this activity, processing and removing log files.

We use a write-ahead-provenance (WAP) protocol to ensure that on-disk provenance accurately reflects on-disk data. WAP is analogous to database write-ahead logging. Enforcing WAP requires that all provenance records be written to disk before the data they describe. This eliminates the possibility that unprovenanced data exists on the disk. In addition, we use transactional structures in the log along with MD5sums of data so that during file system recovery, we identify any data for which the provenance is inconsistent. This indicates precisely the data that was being written to disk at the time of a crash. Thus, Lasagna’s DPAPI interface along with the WAP protocol ensures that provenance is consistent with the data it describes (or, after a crash, inconsistencies are identified).

5.7 Querying

Most existing provenance systems use either an XML-based or a relational representation. We found both lacking. XML has a notion of paths (XPath) but is inherently tree-structured and does not extend well to graphs. SQL has no native concept of paths; writing path-like queries in SQL requires mentally translating the paths into recursive queries, which are themselves expensive and unnatural in a relational environment. It seemed most appropriate to find a query language that was designed specifically for querying graphs.

The Lore semistructured database project at Stanford provided us with the Lorel [1] query language and its “OEM” data model. A semistructured database is one with no fixed schema; the data model in Lore is that of a collection of arbitrary objects, some holding values and some holding tables of named linkages to other objects. The data types of values and linkages are not fixed, and the query language is designed accordingly.

The OEM data model is appealing for provenance, since it naturally represents both graphs and object attributes, and Lorel provides the path-oriented query model for which we were looking. Unfortunately, we found that Lorel had several shortcomings. In particular, it did not support boolean values in the database, its formal grammar was ambiguous, and there were corner cases where the semantics were not well defined. We also needed to extend Lorel to allow traversal of graph edges in both directions. We present a more in-depth discussion of these issues in a recent publication [16].

We developed a new query language based on Lorel, which we call Path Query Language (PQL or “pickle”). It is specifically geared to handle our requirements for

querying provenance. PQL’s query model is based on following paths through an object graph to find and retrieve data. The typical query returns a set of values. The general structure of a PQL query is: **select outputs from sources where condition**. Sources are path expressions, which represent paths through the graph, outputs are anything we can compute on paths, and conditions are boolean predicates like in a SQL query. The PQL reference manual is available online [15].

The following sample query determines the cause of the anomaly in the output in the use case described in Section 3.1.

```
select Ancestor
from Provenance.file as Atlas
    Atlas.input* as Ancestor
where Atlas.name = "atlas-x.gif"
```

The query returns all the ancestors of one output file, `atlas-x.gif` (by following zero or more `input` relationships), which will include both the Kepler workflow entities and the PASS data for the input files. PQL queries, if not posed carefully, can result in information overload. Pruning the query results to produce more focused results is an area of ongoing research.

6 Provenance-Aware Applications

The following sections present technical details about how we implemented provenance collection in a variety of different provenance-aware layers, and the provenance we collect in each. We conclude this section with a summary of the lessons learned while constructing these provenance-aware components. Table 1 summarizes the provenance collected by each provenance-aware system.

6.1 Provenance-Aware NFS

We implemented provenance-aware NFS using NFSv4 [26] in Linux 2.6.23.17. Making NFS provenance aware involves addressing two questions: First, while provenance-aware NFS can leverage the PASSv2 analyzer, should that analyzer reside on the client or the server? And second, how do we extend the NFSv4 protocol to support the six DPAPI operations?

6.1.1 Cycles vs. NFS

An analyzer must process all the provenance records at its abstraction layer in order to properly avoid cycles. Consider a process on an NFS client machine accessing data from two different storage servers. The analyzer must reside at the client, because only there is it possible to see all relevant provenance records.

Record Type	Description
PA-NFS	BEGINTXN
	ENDTXN
	FREEZE
PA-Kepler	TYPE
	NAME
	PARAMS
	INPUT
PA-links	TYPE
	VISITED_URL
	FILE_URL
	CURRENT_URL
	INPUT
PA-Python	TYPE
	NAME
	INPUT

Table 1: Provenance records collected by each provenance-aware application.

Next, consider two programs running on different clients accessing a single server. By the same logic, the analyzer must reside on the server, because only there can it see all related provenance records.

Finally, combine these two scenarios: two client programs each accessing files from two different file servers. In this case, we need analyzers on both clients and servers.

This means that in general we must have an analyzer on every client and also an analyzer on every server; this in turn means that the client instance of the analyzer must be able to stack on top of the server instance, which means that the input and output data representations must be the same. This requirement is easily satisfied as all the components in PASSv2, including the client and the server, communicate via the DPAPI. In fact, it was precisely this observation that motivated layering and the use of the DPAPI as a universal interface.

6.1.2 DPAPI in NFS

pass_write: Supporting `pass_write` requires that we transmit provenance with data to enforce consistency. Accordingly, we created an NFS operation analogous to the local `pass_write`, called `OP_PASSTWRITE`, that transmits both data and provenance to the server. As long as the combined data and provenance size is less than the

NFSv4 client’s block size (typically 64 KB in NFSv4), this approach is sufficient.

Unfortunately, not all data and provenance packets satisfy this constraint. In these cases, we use NFS transactions to encapsulate a collection of operations that must be handled atomically by the server. To support transactions, we introduced two new operations, OP_BEGINTXN and OP_PASSTPROV, and two new provenance record types, BEGINTXN and ENDTXN. First, we invoke an OP_BEGINTXN operation to obtain a transaction ID from the exported PASS volume. We record the transaction ID in a BEGINTXN record at the server. Then, we send the provenance records to the server in 64 KB chunks, using a series of OP_PASSTPROV operations, each identified by the transaction ID acquired by OP_BEGINTXN. Finally, we invoke an OP_PASSTWRITE operation that transmits the data along with a single ENDTXN record. The ENDTXN record contains the transaction ID obtained in OP_BEGINTXN and signals the end of that transaction. A corresponding ENDTXN record is written to the log at the server.

We considered an alternate implementation that obtains a mandatory lock on the file, writes the provenance, and then writes the data as a separate operation. This approach would have provided the coupling between provenance and data; however, it does not allow us to recover from a client crash. If the client wrote the provenance, crashed before sending the data, and then came back up, there is no way for the server to determine that the provenance must be discarded. Our implementation solves this problem, because the transaction ID enables the server’s Waldo daemon to identify the orphaned provenance.

pass_read: For NFS pass_read, we introduced a new operation OP_PASSTREAD, which returns both the requested data and its pnode number and version.

pass_freeze: We send pass_freeze operations to the server as a provenance record type in OP_PASSTWRITE. When the analyzer at the client issues a pass_freeze, the client increments the version locally and attaches a freeze record to the file. The client can then return the correct version of the file on a pass_read without a trip to the server. Later, when the client sends the file’s provenance to the server with an OP_PASSTWRITE, the server processes the freeze records, incrementing its version number accordingly.

We implement freeze as a record type instead of an operation because operations may arrive at the server out of order. pass_freeze is order-sensitive with respect to pass_write. pass_freeze breaks cycles in the records that are about to be written with a pass_write and an out of order arrival can result in a failure to break cycles. Making pass_freeze a record type couples pass_freeze with pass_write and

avoids the problem.

Due to the close-to-open consistency model that NFS supports, two different clients can open the same version of a file and concurrently make modifications to it. Hence, our approach of versioning at the client and updating versions at the server can lead to *version branching*, where two clients create independent copies of an object with the same version. This has not caused any problems in our existing applications, and given the overall lack of precise consistency semantics in NFS, we do not expect it to be problematic for existing applications.

pass_mkobj: We added a new operation called OP_PASSTMKOBJ that returns a unique pnode referencing the object in future interactions. The client then constructs an in-memory anonymous inode that has a reference to the pnode and exports the inode to user-level as a file.

We could have implemented pass_mkobj by creating a file handle at the server and returning it to the client. The client would then use the handle to write provenance. However, this approach would make it difficult to recover from either a server or client crash. The advantage of our approach is that the server only needs enough state to verify that the pnode is a valid on pass_reviveobj and requires no complicated recovery. If the server crashes and comes back up, the client can continue to use the pnode as though the crash never happened, as the pnode is just a number. Similarly, if the client crashes, the server does not have to clean up state as it has only allocated a (cheap) pnode number to the client.

pass_reviveobj: We added a new operation called OP_PASSTREVIVEOBJ that verifies that the given pnode number is valid and returns an anonymous inode as we do for pass_mkobj.

pass_sync: This is implemented by invoking the OP_PASSTPROV operation. When the provenance exceeds 64KB, we encapsulate the operation in a transaction as we do for pass_write.

6.2 Provenance-Aware Kepler

Kepler records provenance for all communication between workflow operators, recording these events either in a text file or relational database. We added a third recording option: transmitting the provenance into PASSv2 via the DPAPI. This integration was simple. We implemented methods in Kepler’s provenance recording interface that translate Kepler’s provenance events into explicit ancestor-descendant relationships.

We create a PASS object for every workflow operator using pass_mkobj and set its properties, such as NAME, TYPE, and PARAMS, which specify the names and values of its parameters (such as “fileName” or “confirmOverwrite” for a file output operator). When an op-

erator produces a result, Kepler notifies our recording interface with its event mechanism. Upon receipt of the event, we add an ancestry relationship between this operator and every recipient of the message by issuing a `pass_write` call that records the ancestry between the sender and the recipient. This is the only one of Kepler's recording operations that needs to send data to PASSv2.

Unfortunately, the recording interface does not provide methods to generate provenance for reading or writing files or downloading data from the Internet. Instead, Kepler knows about data sink and source operators, which open and close files. We modified the Kepler routines used by these operators to infer the files that are being read/written, linking Kepler's provenance to that in PASSv2.

6.3 Provenance-Aware links

We chose to add provenance collection to version 0.98 of `links`, a text-based browser, as it had the simplest code base of those browsers we examined. We are currently exploring provenance collection in a Firefox [19].

A PA-browser can capture semantic information that is invisible to PASS, such as:

- The URL of any file that a user downloads using the browser;
- The web page a user was examining when she initiated a download;
- The sequence of web pages a user visited before downloading a file; and
- The set of pages that were active concurrently.

We group provenance by session, as it represents a logical task performed by a user. On session creation, we create a PASS object that represents it (using `pass_mkobj`) and record the object `TYPE` (using `pass_write`). Whenever a user visits a site, we generate a `VISITED_URL` record that describes the dependency between the session and the URL and record it by issuing the DPAPI call `pass_write`. These records identify the sequence of URLs that a user visited before downloading a file.

Each time the browser downloads a file, we generate three records. An `INPUT` record captures the dependency between the file and the session, connecting the file to the sequence of URLs visited during the session, before initiating the download. A `FILE_URL` record captures the URL of the file itself. A `CURRENT_URL` record captures the dependency between the file and the page the user was viewing when she decided to download the file. We replace the `write` that the browser issues to record the file on disk with a `pass_write` that transmits the data and the three provenance records to PASSv2.

6.4 Provenance-Aware Python Apps

We discovered that a colleague had written a set of wrappers to track provenance in Python applications. His goal was to explicitly identify relationships between input and output files using Python scripts that read in a large number of data files, but used only a subset of them.

To make the Python analysis program provenance-aware, we created Python bindings for our DPAPI interface. We also wrap objects, modules, basic types, and output files with code that creates PASSv2 objects representing our Python objects (using `pass_mkobj`), intercepts method invocations, and then records the relationships between the objects. By wrapping a few modules and objects we record the information flow pertaining to those objects and methods and relate them to the files they eventually affect. For every object, we record the object `TYPE` (for example, `FUNCTION`) and the object `NAME`. For modules and methods, we add an intercept for each method so we can connect method invocations to their inputs and outputs. On every method invocation, we issue DPAPI `pass_write` calls to record `INPUT` records describing the dependencies between each input and its method invocation and between the method invocation and each of its outputs.

6.5 Summary and Lessons Learned

While provenance-aware applications are generally useful and many developers develop *ad hoc* solutions to the problems they solve, the ability to integrate such solutions with system-level approaches increases the value of both the system-level provenance and the application-level provenance. Our system has a simple architecture and API that enables such an integration. We now discuss some of the lessons we learned.

Our experience with `links`, Kepler, and Python led us to the following guidelines for making applications provenance-aware. First, application developers have to identify the provenance they want to collect. Next, they have to replace read calls with `pass_read` calls and write calls with `pass_write` calls, obtaining and forwarding provenance to the layers around them. In order to record semantic provenance, application developers can create objects using `pass_mkobj` and record such provenance via `pass_write` calls on those objects. They can then link the semantic provenance with the system objects by creating appropriate records and storing them via `pass_write` calls. Finally, layers that are a substrate to higher level applications (like an interpreter) must export the DPAPI. If they do not export the DPAPI, the applications cannot layer provenance on top of them.

It is not trivial to extend existing complex applications,

which were not designed to collect provenance, to make them provenance aware. We observe this in our ongoing work with Firefox. In Firefox, interesting provenance events such as page loads, bookmarks, etc. occur in the user interface modules. However, the I/O manipulation events such as cache and file writes occur in completely different modules. Connecting provenance collected in the UI modules to data writes in the I/O modules entails a significant amount of re-engineering of Firefox modules and interfaces. We are currently working on this.

Considering that operating systems are, to this day, introducing new system calls, we expect the DPAPI to evolve over time. It has continued to evolve over the course of the project and this paper. As we noted in Section 5.2, we initially designed the objects returned by `pass_mkobj` to be transient. However, while working on Firefox provenance collection, we discovered that we needed to revive these objects. For example, in Firefox, we create an object per active session. Firefox stores the sessions to disk and restores them when the user restarts the browser. In this scenario, the application needs to revive the objects used to record each session’s provenance so as to record further provenance. Hence, in order to support such scenarios, we extended the DPAPI to include `pass_reviveobj`.

We initially believed that the Python wrappers we built were sufficient to enable provenance-aware Python applications. We later realized that while we could wrap functions, we lost provenance across built-in operators. In retrospect, what we discovered with Python was the difference between building a provenance-aware system and provenance-aware applications. By wrapping function calls in Python, we make an application provenance-aware, as we did for `links` and `Kepler`. Making Python itself provenance-aware would require modifying the Python interpreter, as we modified the operating system to make it provenance-aware. While an interesting project, we have left that undertaking for future research.

7 Performance Evaluation

While the main contribution of this work is in the new capabilities available from the system, we wanted to verify that these capabilities do not impose excessive overheads. There are two concerns: the execution time overhead due to the additional work done to collect provenance and the space overhead for storing provenance.

We evaluate these overheads using five applications representative of a broad range of workloads: 1) Linux compile, in which we unpack and build Linux kernel version 2.6.19.1. This represents a CPU intensive workload; 2) Postmark, that simulates the operation of an email server. We ran 1500 transactions with file sizes ranging from 4 KB to 1 MB, with 10 subdirectories and 1500

files. This benchmark is representative of an I/O intensive workload; 3) Mercurial activity benchmark, where we evaluate the overhead a user experiences in a normal development scenario. We start with a vanilla Linux 2.6.19.1 kernel and apply, as patches, each of the changes that we committed to our own Mercurial-managed source tree; 4) Blast, a biological workload used to find protein sequences in a species that are closely related to the protein sequences in another species. The workload formats two input data files with a tool called `formatdb`, then processes the two files with `Blast`, and then massages the output data with a series of Perl scripts; and 5) A PA-Kepler workload, that parses tabular data, extracts values, and reformats it using a user-specified expression. The PA-Kepler workload, when located on a PA-NFS volume, is similar to the situation presented in Section 1, where provenance collection is integrated across three layers.

We ran two batches of experiments: one comparing PASSv2 to vanilla ext3 (in ordered mode) and another on comparing provenance-aware NFS (PA-NFS) to NFS exporting ext3 (also in ordered mode). We ran all local benchmarks on a 3GHz Pentium 4 machine with 512MB of RAM, an 80GB 7200 RPM Western Digital Caviar WD800JB hard drive and with a kernel (Vanilla/PASS) based on Linux 2.6.23.17. For experiments involving NFS, we used the previous machine as the server and a 2.8GHz 2 CPU Opteron 254 machine with 3GB of RAM as the client. The client machine has the same software configuration as the server.

PASSv2 Elapsed Time Results: Table 2 shows the elapsed time overheads. The general pattern we observed is that the elapsed times are affected when provenance writes interfere with the workload’s regular writes. The Mercurial activity benchmark has the highest elapsed time overhead of 23.1% despite having minimal space overhead. This is because `patch` performs many metadata operations (it creates a temporary file, merges data from the patch file and the original file into the temporary file, and finally renames the temporary file). The provenance writes interfere with `patch`’s metadata I/O, leading to extra seeks, which increase overhead. The Linux kernel compile has an overhead of 15.6% due to provenance writes. Postmark has an overhead of 11.5%, and the overheads, unlike the former two benchmarks, are due to the double buffering in Lasagna (stackable file systems cache both their data pages and lower file system data pages). Blast and PA-Kepler are heavily CPU bound and hence their elapsed time are minimally affected due to provenance writes.

PA-NFS Elapsed Time Results: The PA-NFS elapsed time overheads are lower for Linux compile and Mercurial activity benchmarks compared to PASSv2 overheads, as the additional delay introduced by the net-

Benchmark	Ext3	PASSv2	Overhead	NFS	PA-NFS	Overhead
Linux Compile	1746	2018	15.6%	3320	3353	11.0%
Postmark	453	505	11.5%	636	743	16.8%
Mercurial Activity	614	756	23.1%	2842	3089	8.7%
Blast	69	69.5	0.7%	52	53	1.9%
PA-Kepler	1246	1264	1.4%	160	164	2.5%

Table 2: Elapsed time overheads (in seconds).

Benchmark	Ext3	Provenance	Provenance+Indexes
Linux Compile	1287.9	88.9 (6.9%)	236.8 (18.4%)
Postmark	1289.5	0.8 (0.1%)	1.7 (0.1%)
Mercurial Activity	858.7	15.4 (1.8%)	28.9 (3.4%)
Blast	5.6	0.1 (1.1%)	0.2 (3.8%)
PA-Kepler	3.5	0.2 (4.7%)	0.5 (14.2%)

Table 3: Space overheads (in MB) for PASSv2. The space overheads for PA-NFS are similar.

work round trips affect both NFS and PA-NFS equally. The Postmark overheads, though reasonable, are higher in PA-NFS compared to PASSv2. Our experiments confirm that out of 16.8% overhead that Postmark incurs for PA-NFS, 14.8% is due to the fact that Lasagna is implemented as a stackable file system. The Blast and PA-Kepler overheads remain minimal even in the PA-NFS case.

Space Overheads: Table 3 shows the provenance database space overhead and the total space overheads (provenance database and indexes) for PASSv2. The overheads are computed as a percentage of the Ext3 space utilization. Overall, the provenance database overheads are minimal with all overheads being less than 7%. The total space overheads are reasonable with Linux compile having the highest overhead at 18.4%. PA-Kepler combines both system provenance and application provenance and has a total space overhead of 14.2%. For the rest of the benchmarks, the space overhead is less than 4%. The space overheads for PA-NFS as the overheads are similar to the overheads in PASSv2.

8 Related Work

Several systems have looked at propagating taint information along with the data in order to debug applications or to detect security violations [22, 30]. These systems are, however, extremely slow as they track information flow at a fine granularity and hence can never be used in production systems. PASSv2 monitors only system call events, which is much less expensive. The drawback is that the information collected by PASSv2 can be less accurate; but as we have shown in the use cases, it is valuable nonetheless.

X-Trace [7] is a research prototype built to diagnose

problems in network applications that span multiple protocol layers and administrative domains. X-Trace’s approach is similar to ours in that it integrates information from multiple layers in the system stack. The higher layer generates a “taint” that is propagated through the layers along with the data. The generated debug metadata is not sent with the data, but is instead sent out of band to a destination. Hence the interface between layers can be much more limited compared to the DPAPI in PASSv2. Furthermore, X-Trace does not need to deal with cycles as the PASSv2 analyzer does.

Another class of systems that maintain dependencies are software build systems such as Vesta [14]. These systems need the initial dependencies be specified manually. Build systems maintain dependencies after those dependencies have been specified; PASS derives dependencies based upon program execution. As a result, while extraordinarily useful for software development, they ignore the central PASS challenge: automatically generating the derivation rules as a system runs.

Chanda et. al. [5] present a mechanism to use causal information flow to introduce new functionality. For example, one can send process priority along with data to a socket. On receiving the data and the causal metadata (priority), the server increases the priority for processing this data. This mechanism is complementary to the ideas we have explored in this work.

9 Conclusions

We have presented a provenance-aware storage system that permits integration of provenance across multiple layers of abstraction, ranging from Python applications to network-attached storage. This integration requires a layered architecture that dictates how provenance, data,

and versions must flow through the system. The architecture has proved versatile enough to facilitate integration with a variety of applications and NFS, providing functionality not available in systems that cannot integrate provenance across different layers of abstraction.

We have presented several use cases illustrating what kinds of functionality this layering enables. The use cases show efficacy in a variety of areas, such as malware tracking and scientific data processing. Finally, we demonstrated an end-to-end system encompassing provenance-aware applications and network-attached storage, imposing reasonable space and time overheads ranging between 1% and 23%.

Acknowledgments: We thank Stephen D. Holland for providing us the PA-Python use cases and Joseph Barrillari for help with PA-Python development. We thank Andrew Warfield, our shepherd, for repeated careful and thoughtful reviews of our paper. We thank Keith Bostic, Stephen D. Holland, Keith Smith, and Jonathan Ledlie for their feedback on early drafts of the paper. We thank the anonymous reviewers for the valuable feedback they provided. This work was partially made possible thanks to NSF grants CNS-0614784 and IIS-0849392.

References

- [1] ABITEBOUL, S., QUASS, D., MCHUGH, J., WIDOM, J., AND WIENER, J. L. The Lorel query language for semistructured data. *International Journal on Digital Libraries* 1, 1 (1997), 68–88.
- [2] ALTINTAS, I., BARNEY, O., AND JAEGER-FRANK, E. Provenance collection support in the Kepler scientific workflow system. In *IPAW* (2006), vol. 4145 of *LNCS*, Springer.
- [3] BOSE, R., AND FREW, J. Composing lineage metadata with xml for custom satellite-derived data products. In *Proceedings of the Sixteenth International Conference on Scientific and Statistical Database Management* (2004).
- [4] BRAUN, U., SHINNAR, A., AND SELTZER, M. Securing Provenance. In *Proceedings of HotSec 2008* (July 2008).
- [5] CHANDA, A., ELMELEEGY, K., COX, A. L., AND ZWAENEPOEL, W. Causeway: operating system support for controlling and analyzing the execution of distributed programs. In *HOTOS* (2005).
- [6] EFSTATOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *SOSP* (2005).
- [7] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *In NSDI* (2007).
- [8] FOSTER, I., AND KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing* (Summer 1997).
- [9] FOSTER, I., VOECKLER, J., WILDE, M., AND ZHAO, Y. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR* (Asilomar, CA, Jan. 2003).
- [10] GenePattern. <http://www.broad.mit.edu/cancer/software/genepattern>.
- [11] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The Taser intrusion recovery system. In *SOSP* (2005).
- [12] HALCROW, M. A. eCryptfs: An enterprise-class encrypted filesystem for linux. *Ottawa Linux Symposium* (2005).
- [13] HASAN, R., SION, R., AND WINSLETT, M. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *FAST* (2009).
- [14] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. The Vesta Approach to Software Configuration Management. Technical Report 168, Compaq Systems Research Center, March 2001.
- [15] HOLLAND, D. A. PQL language guide and reference. <http://www.eecs.harvard.edu/syrah/pql/docs/>. Harvard University, 2009.
- [16] HOLLAND, D. A., BRAUN, U., MACLEAN, D., MUNISWAMY-REDDY, K.-K., AND SELTZER, M. I. A Data Model and Query Language Suitable for Provenance. In *Proceedings of the 2008 International Provenance and Annotation Workshop (IPA)*.
- [17] KING, S. T., AND CHEN, P. M. Backtracking Intrusions. In *SOSP* (Bolton Landing, NY, October 2003).
- [18] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *symposium on Operating systems principles* (2007).
- [19] MARGO, D. W., AND SELTZER, M. The case for browser provenance. In *1st Workshop on the Theory and Practice of Provenance* (2009).
- [20] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-Based Versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (Feb 2009).
- [21] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*.
- [22] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS* (2005).
- [23] Provenance aware service oriented architecture. <http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome>.
- [24] The First Provenance Challenge. <http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge>.
- [25] The Second Provenance Challenge. <http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge>.
- [26] SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., BEAME, C., EISLER, M., AND NOVEC, D. Network File System (NFS) version 4 Protocol. <http://www.ietf.org/rfc/rfc3530.txt>, April 2003.
- [27] SUNDARAMAN, S., SIVATHANU, G., AND ZADOK, E. Selective versioning in a secure disk system. In *Proceedings of the 17th USENIX Security Symposium* (July-August 2008).
- [28] VAHDAT, A., AND ANDERSON, T. Transparent result caching. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference* (1998).
- [29] WIDOM, J. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR* (Asilomar, CA, January 2005).
- [30] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS* (2007).
- [31] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *OSDI* (2006).

Object Storage on CRAQ

High-throughput chain replication for read-mostly workloads

Jeff Terrace and Michael J. Freedman
Princeton University

Abstract

Massive storage systems typically replicate and partition data over many potentially-faulty components to provide both reliability and scalability. Yet many commercially-deployed systems, especially those designed for interactive use by customers, sacrifice stronger consistency properties in the desire for greater availability and higher throughput.

This paper describes the design, implementation, and evaluation of CRAQ, a distributed object-storage system that challenges this inflexible tradeoff. Our basic approach, an improvement on Chain Replication, maintains strong consistency while greatly improving read throughput. By distributing load across all object replicas, CRAQ scales linearly with chain size without increasing consistency coordination. At the same time, it exposes non-committed operations for weaker consistency guarantees when this suffices for some applications, which is especially useful under periods of high system churn. This paper explores additional design and implementation considerations for geo-replicated CRAQ storage across multiple datacenters to provide locality-optimized operations. We also discuss multi-object atomic updates and multicast optimizations for large-object updates.

1 Introduction

Many online services require *object-based* storage, where data is presented to applications as entire units. Object stores support two basic primitives: *read* (or query) operations return the data block stored under an object name, and *write* (or update) operations change the state of a single object. Such object-based storage is supported by key-value databases (*e.g.*, BerkeleyDB [40] or Apache’s semi-structured CouchDB [13]) to the massively-scalable systems being deployed in commercial datacenters (*e.g.*, Amazon’s Dynamo [15], Facebook’s Cassandra [16], and the popular Memcached [18]). To achieve the requisite reliability, load balancing, and scalability in many of these systems, the object namespace is partitioned over many machines and each data object is replicated several times.

Object-based systems are more attractive than their file-system counterparts when applications have certain requirements. Object stores are better suited for flat namespaces, such as in key-value databases, as opposed to hierarchical directory structures. Object stores simplify the process of supporting whole-object modifications. And, they typically only need to reason about the ordering of modifications *to a specific object*, as opposed to the entire storage system; it is significantly cheaper to provide consistency guarantees per object instead of across all operations and/or objects.

When building storage systems that underlie their myriad applications, commercial sites place the need for high performance and availability at the forefront. Data is replicated to withstand the failure of individual nodes or even entire datacenters, whether from planned maintenance or unplanned failure. Indeed, the news media is rife with examples of datacenters going offline, taking down entire websites in the process [26]. This strong focus on availability and performance—especially as such properties are being codified in tight SLA requirements [4, 24]—has caused many commercial systems to sacrifice *strong consistency* semantics due to their perceived costs (as at Google [22], Amazon [15], eBay [46], and Facebook [44], among others).

Recently, van Renesse and Schneider presented a *chain replication* method for object storage [47] over fail-stop servers, designed to provide strong consistency yet improve throughput. The basic approach organizes all nodes storing an object in a chain, where the chain tail handles all read requests, and the chain head handles all write requests. Writes propagate down the chain before the client is acknowledged, thus providing a simple ordering of all object operations—and hence strong consistency—at the tail. The lack of any complex or multi-round protocols yields simplicity, good throughput, and easy recovery.

Unfortunately, the basic chain replication approach has some limitations. All reads for an object must go to the same node, leading to potential hotspots. Multiple chains can be constructed across a cluster of nodes for better load balancing—via consistent hashing [29] or a more centralized directory approach [22]—but these algorithms might

still find load imbalances if particular objects are disproportionately popular, a real issue in practice [17]. Perhaps an even more serious issue arises when attempting to build chains across multiple datacenters, as all reads to a chain may then be handled by a potentially-distant node (the chain’s tail).

This paper presents the design, implementation, and evaluation of CRAQ (*Chain Replication with Apportioned Queries*), an object storage system that, while maintaining the strong consistency properties of chain replication [47], provides lower latency and higher throughput for read operations by supporting *apportioned queries*: that is, dividing read operations over all nodes in a chain, as opposed to requiring that they all be handled by a single primary node. This paper’s main contributions are the following.

1. CRAQ enables any chain node to handle read operations while preserving strong consistency, thus supporting load balancing across all nodes storing an object. Furthermore, when workloads are read mostly—an assumption used in other systems such as the Google File System [22] and Memcached [18]—the performance of CRAQ rivals systems offering only eventual consistency.
2. In addition to strong consistency, CRAQ’s design naturally supports eventual-consistency among read operations for lower-latency reads during write contention and degradation to read-only behavior during transient partitions. CRAQ allows applications to specify the maximum staleness acceptable for read operations.
3. Leveraging these load-balancing properties, we describe a wide-area system design for building CRAQ chains across geographically-diverse clusters that preserves strong locality properties. Specifically, reads can be handled either completely by a local cluster, or at worst, require concise metadata information to be transmitted across the wide-area during times of high write contention. We also present our use of ZooKeeper [48], a PAXOS-like group membership system, to manage these deployments.

Finally, we discuss additional extensions to CRAQ, including the integration of mini-transactions for multi-object atomic updates, and the use of multicast to improve write performance for large-object updates. We have not yet finished implementing these optimizations, however.

A preliminary performance evaluation of CRAQ demonstrates its high throughput compared to the basic chain replication approach, scaling linearly with the number of chain nodes for read-mostly workloads: approximately a 200% improvement for three-node chains, and 600% for seven-node chains. During high write contention, CRAQ’s read throughput in three-node chains still

outperformed chain replication by a factor of two, and read latency remains low. We characterize its performance under varying workloads and under failures. Finally, we evaluate CRAQ’s performance for geo-replicated storage, demonstrating significantly lower latency than that achieved by basic chain replication.

The remainder of this paper is organized as follows. Section §2 provides a comparison between the basic chain replication and CRAQ protocols, as well as CRAQ’s support for eventual consistency. Section §3 describes scaling out CRAQ to many chains, within and across datacenters, as well as the group membership service that manages chains and nodes. Section §4 touches on extensions such as multi-object updates and leveraging multicast. Section §5 describes our CRAQ implementation, §6 presents our performance evaluation, §7 reviews related work, and §8 concludes.

2 Basic System Model

This section introduces our object-based interface and consistency models, provides a brief overview of the standard Chain Replication model, and then presents strongly-consistent CRAQ and its weaker variants.

2.1 Interface and Consistency Model

An object-based storage system provides two simple primitives for users:

- ***write(objID, V)***: The write (update) operation stores the value V associated with object identifier $objID$.
- ***V ← read(objID)***: The read (query) operation retrieves the value V associated with object id $objID$.

We will be discussing two main types of consistency, taken with respect to individual objects.

- **Strong Consistency** in our system provides the guarantee that all read and write operations to an object are executed in some sequential order, and that a read to an object always sees the latest written value.
- **Eventual Consistency** in our system implies that writes to an object are still applied in a sequential order on all nodes, but eventually-consistent reads to different nodes can return stale data for some period of inconsistency (*i.e.*, before writes are applied on all nodes). Once all replicas receive the write, however, read operations will never return an older version than this latest committed write. In fact, a client will also see monotonic read consistency¹ if it main-

¹That is, informally, successive reads to an object will return either the same prior value or a more recent one, but never an older value.

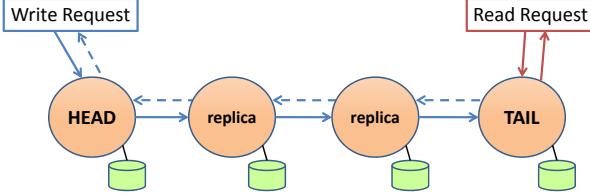


Figure 1: All reads in Chain Replication must be handled by the tail node, while all writes propagate down the chain from the head.

tains a session with a particular node (although not across sessions with different nodes).

We next consider how Chain Replication and CRAQ provide their strong consistency guarantees.

2.2 Chain Replication

Chain Replication (CR) is a method for replicating data across multiple nodes that provides a strongly consistent storage interface. Nodes form a *chain* of some defined length C . The *head* of the chain handles all *write* operations from clients. When a write operation is received by a node, it is propagated to the next node in the chain. Once the write reaches the tail node, it has been applied to all replicas in the chain, and it is considered *committed*. The tail node handles all *read* operations, so only values which are committed can be returned by a *read*.

Figure 1 provides an example chain of length four. All read requests arrive and are processed at the tail. Write requests arrive at the head of the chain and propagate their way down to the tail. When the tail commits the write, a reply is sent to the client. The CR paper describes the tail sending a message directly back to the client; because we use TCP, our implementation actually has the head respond after it receives an acknowledgment from the tail, given its pre-existing network connection with the client. This acknowledgment propagation is shown with the dashed line in the figure.

The simple topology of CR makes write operations cheaper than in other protocols offering strong consistency. Multiple concurrent writes can be pipelined down the chain, with transmission costs equally spread over all nodes. The simulation results of previous work [47] showed competitive or superior throughput for CR compared to primary/backup replication, while arguing a principle advantage from quicker and easier recovery.

Chain replication achieves strong consistency: As all reads go to the tail, and all writes are committed only when they reach the tail, the chain tail can trivially apply a total ordering over all operations. This does come at a cost, however, as it reduces read throughput to that of a single node, instead of being able to scale out with chain

size. But it is necessary, as querying intermediate nodes could otherwise violate the strong consistency guarantee; specifically, concurrent reads to different nodes could see different writes as they are in the process of propagating down the chain.

While CR focused on providing a storage service, one could also view its query/update protocols as an interface to replicated state machines (albeit ones that affect distinct object). One can view CRAQ in a similar light, although the remainder of this paper considers the problem only from the perspective of a read/write (also referred to as a get/put or query/update) object storage interface.

2.3 Chain Replication with Apportioned Queries

Motivated by the popularity of read-mostly workload environments, CRAQ seeks to increase read throughput by allowing any node in the chain to handle read operations while still providing strong consistency guarantees. The main CRAQ extensions are as follows.

1. A node in CRAQ can store multiple versions of an object, each including a monotonically-increasing version number and an additional attribute whether the version is *clean* or *dirty*. All versions are initially marked as clean.
2. When a node receives a new version of an object (via a write being propagated down the chain), the node appends this latest version to its list for the object.
 - If the node is not the tail, it marks the version as dirty, and propagates the write to its successor.
 - Otherwise, if the node is the tail, it marks the version as clean, at which time we call the object version (write) as *committed*. The tail node can then notify all other nodes of the commit by sending an acknowledgement backwards through the chain.
3. When an *acknowledgment* message for an object version arrives at a node, the node marks the object version as clean. The node can then delete all prior versions of the object.
4. When a node receives a read request for an object:
 - If the latest known version of the requested object is clean, the node returns this value.
 - Otherwise, if the latest version number of the object requested is dirty, the node contacts the tail and asks for the tail's last committed version number (*a version query*). The node then returns that version of the object; by construction, the node is guaranteed to be storing this

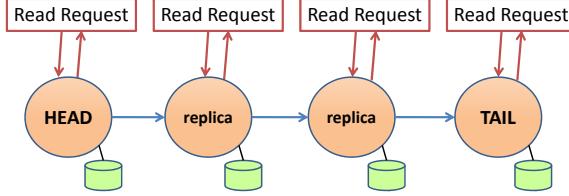


Figure 2: Reads to clean objects in CRAQ can be completely handled by any node in the system.

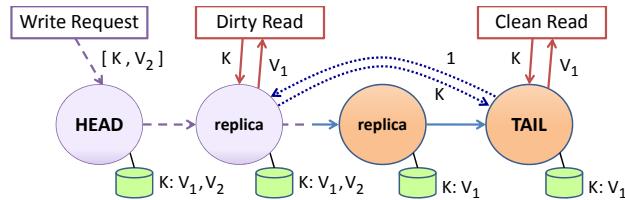


Figure 3: Reads to dirty objects in CRAQ can be received by any node, but require small version requests (dotted blue line) to the chain tail to properly serialize operations.

version of the object. We note that although the tail could commit a new version between when it replied to the version request and when the intermediate node sends a reply to the client, this does not violate our definition of strong consistency, as *read operations are serialized with respect to the tail*.

Note that an object’s “dirty” or “clean” state at a node can also be determined implicitly, provided a node deletes old versions as soon as it receives a write commitment acknowledgment. Namely, if the node has exactly one version for an object, the object is implicitly in the clean state; otherwise, the object is dirty and the properly-ordered version must be retrieved from the chain tail.

Figure 2 shows a CRAQ chain in the starting clean state. Each node stores an identical copy of an object, so any read request arriving at any node in the chain will return the same value. All nodes remain in the clean state unless a write operation is received.²

In Figure 3, we show a write operation in the middle of propagation (shown by the dashed purple line). The head node received the initial message to write a new version (V_2) of the object, so the head’s object is dirty. It then propagated the write message down the chain to the sec-

²There’s a small caveat about the system ordering properties for clean reads. In traditional Chain Replication, all operations are handled by the tail, so it explicitly defines a total ordering over all operations affecting an object. In CRAQ, clean read operations to different nodes are executed locally; thus, while one could define an (arbitrary) total ordering over these “concurrent” reads, the system does not do such explicitly. Of course, both systems explicitly maintain (at the tail) a total ordering with respect to all read/write, write/read, and write/write relationships.

ond node, which also marked itself as dirty for that object (having multiple versions [V_1, V_2] for a single object ID K). If a read request is received by one of the clean nodes, they immediately return the old version of the object: This is correct, as the new version has yet to be committed at the tail. If a read request is received by either of the dirty nodes, however, they send a version query to the tail—shown in the figure by the dotted blue arrow—which returns its known version number for the requested object (1). The dirty node then returns the old object value (V_1) associated with this specified version number. Therefore, all nodes in the chain will still return the same version of an object, even in the face of multiple outstanding writes being propagated down the chain.

When the tail receives and accepts the write request, it sends an acknowledgment message containing this write’s version number back up the chain. As each predecessor receives the acknowledgment, it marks the specified version as clean (possibly deleting all older versions). When its latest-known version becomes clean, it can subsequently handle reads locally. This method leverages the fact that writes are all propagated serially, so the tail is always the last chain node to receive a write.

CRAQ’s throughput improvements over CR arise in two different scenarios:

- **Read-Mostly Workloads** have most of the read requests handled solely by the $C - 1$ non-tail nodes (as clean reads), and thus throughput in these scenarios scales linearly with chain size C .
- **Write-Heavy Workloads** have most read requests to non-tail nodes as dirty, thus require version queries to the tail. We suggest, however, that these version queries are lighter-weight than full reads, allowing the tail to process them at a much higher rate before it becomes saturated. This leads to a total read throughput that is still higher than CR.

Performance results in §6 support both of these claims, even for small objects. For longer chains that are persistently write-heavy, one could imagine optimizing read throughput by having the tail node *only* handle version queries, not full read requests, although we do not evaluate this optimization.

2.4 Consistency Models on CRAQ

Some applications may be able to function with weaker consistency guarantees, and they may seek to avoid the performance overhead of version queries (which can be significant in wide-area deployments, per §3.3), or they may wish to continue to function at times when the system cannot offer strong consistency (e.g., during partitions). To support such variability in requirements, CRAQ simultaneously supports three different consistency models for

reads. A read operation is annotated with which type of consistency is permissive.

- **Strong Consistency** (the default) is described in the model above (§2.1). All object reads are guaranteed to be consistent with the last committed write.
- **Eventual Consistency** allows read operations to a chain node to return the newest object version known to it. Thus, a subsequent read operation to a different node may return an object version older than the one previously returned. This does not, therefore, satisfy monotonic read consistency, although reads to a single chain node do maintain this property locally (*i.e.*, as part of a *session*).
- **Eventual Consistency with Maximum-Bounded Inconsistency** allows read operations to return newly written objects before they commit, but only to a certain point. The limit imposed can be based on time (relative to a node’s local clock) or on absolute version numbers. In this model, a value returned from a read operation is guaranteed to have a maximum inconsistency period (defined over time or versioning). If the chain is still available, this inconsistency is actually in terms of the returned version being *newer* than the last committed one. If the system is partitioned and the node cannot participate in writes, the version may be *older* than the current committed one.

2.5 Failure Recovery in CRAQ

As the basic structure of CRAQ is similar to CR, CRAQ uses the same techniques to recover from failure. Informally, each chain node needs to know its predecessor and successor, as well as the chain head and tail. When a head fails, its immediate successor takes over as the new chain head; likewise, the tail’s predecessor takes over when the tail fails. Nodes joining or failing from within the middle of the chain must insert themselves between two nodes, much like a doubly-linked list. The proofs of correctness for dealing with system failures are similar to CR; we avoid them here due to space limitations. Section §5 describes the details of failure recovery in CRAQ, as well as the integration of our coordination service. In particular, CRAQ’s choice of allowing a node to join anywhere in a chain (as opposed only to at its tail [47]), as well as properly handling failures during recovery, requires some careful consideration.

3 Scaling CRAQ

In this section, we discuss how applications can specify various chain layout schemes in CRAQ, both within a single datacenter and across multiple datacenters. We then

describe how to use a coordination service to store the chain metadata and group membership information.

3.1 Chain Placement Strategies

Applications that use distributed storage services can be diverse in their requirements. Some common situations that occur may include:

- Most or all writes to an object might originate in a single datacenter.
- Some objects may be only relevant to a subset of datacenters.
- Popular objects might need to be heavily replicated while unpopular ones can be scarce.

CRAQ provides flexible chain configuration strategies that satisfy these varying requirements through the use of a two-level naming hierarchy for objects. An object’s identifier consists of both a *chain identifier* and a *key identifier*. The chain identifier determines which nodes in CRAQ will store all keys within that chain, while the key identifier provides unique naming per chain. We describe multiple ways of specifying application requirements:

1. Implicit Datacenters & Global Chain Size:

$$\{num_datacenters, chain_size\}$$

In this method, the number of datacenters that will store the chain is defined, but not explicitly which datacenters. To determine exactly which datacenters store the chain, consistent hashing is used with unique datacenter identifiers.

2. Explicit Datacenters & Global Chain Size:

$$\{chain_size, dc_1, dc_2, \dots, dc_N\}$$

Using this method, every datacenter uses the same chain size to store replicas within the datacenter. The head of the chain is located within datacenter dc_1 , the tail of the chain is located within datacenter dc_N , and the chain is ordered based on the provided list of datacenters. To determine which nodes within a datacenter store objects assigned to the chain, consistent hashing is used on the chain identifier. Each datacenter dc_i has a node which connects to the tail of datacenter dc_{i-1} and a node which connects to the head of datacenter dc_{i+1} , respectively. An additional enhancement is to allow *chain_size* to be 0 which indicates that the chain should use *all* nodes within each datacenter.

3. Explicit Datacenter Chain Sizes:

$$\{dc_1, chain_size_1, \dots, dc_N, chain_size_N\}$$

Here the chain size within each datacenter is specified separately. This allows for non-uniformity in chain load balancing. The chain nodes within each datacenter are chosen in the same manner as the previous method, and *chain_size* can also be set to 0.

In methods 2 and 3 above, dc_1 can be set as a **master datacenter**. If a datacenter is the master for a chain, this means that writes to the chain will only be accepted by that datacenter during transient failures. Otherwise, if dc_1 is disconnected from the rest of the chain, dc_2 could become the new head and take over write operations until dc_1 comes back online. When a master is not defined, writes will only continue in a partition if the partition contains a majority of the nodes in the global chain. Otherwise, the partition will become read-only for maximum-bounded inconsistent read operations, as defined in Section 2.4.

CRAQ could easily support other more complicated methods of chain configuration. For example, it might be desirable to specify an explicit backup datacenter which only participates in the chain if another datacenter fails. One could also define a set of datacenters (*e.g.*, “East coast”), any one of which could fill a single slot in the ordered list of datacenters of method 2. For brevity, we do not detail more complicated methods.

There is no limit on the number of key identifiers that can be written to a single chain. This allows for highly flexible configuration of chains based on application needs.

3.2 CRAQ within a Datacenter

The choice of how to distribute multiple chains across a datacenter was investigated in the original Chain Replication work. In CRAQ’s current implementation, we place chains within a datacenter using consistent hashing [29, 45], mapping potentially many chain identifiers to a single head node. This is similar to a growing number of datacenter-based object stores [15, 16]. An alternative approach, taken by GFS [22] and promoted in CR [47], is to use the membership management service as a directory service in assigning and storing randomized chain membership, *i.e.*, each chain can include some random set of server nodes. This approach improves the potential for parallel system recovery. It comes at the cost, however, of increased centralization and state. CRAQ could easily use this alternative organizational design as well, but it would require storing more metadata information in the coordination service.

3.3 CRAQ Across Multiple Datacenters

CRAQ’s ability to read from any node improves its latency when chains stretch across the wide-area: When clients

have flexibility in their choice of node, they can choose one that is nearby (or even lightly loaded). As long as the chain is clean, the node can return its local replica of an object without having to send any wide-area requests. With traditional CR, on the other hand, all reads would need to be handled by the potentially-distant tail node. In fact, various designs may choose head and/or tail nodes in a chain based on their datacenter, as objects may experience significant reference locality. Indeed, the design of PNUTS [12], Yahoo!’s new distributed database, is motivated by the high write locality observed in their datacenters.

That said, applications might further optimize the selection of wide-area chains to minimize write latency and reduce network costs. Certainly the naive approach of building chains using consistent hashing across the entire global set of nodes leads to randomized chain successors and predecessors, potentially quite distant. Furthermore, an individual chain may cross in and out of a datacenter (or particular cluster within a datacenter) several times. With our chain optimizations, on the other hand, applications can minimize write latency by carefully selecting the order of datacenters that comprise a chain, and we can ensure that a single chain crosses the network boundary of a datacenter only once in each direction.

Even with an optimized chain, the latency of write operations over wide-area links will increase as more datacenters are added to the chain. Although this increased latency could be significant in comparison to a primary/backup approach which disseminates writes in parallel, it allows writes to be pipelined down the chain. This vastly improves write throughput over the primary/backup approach.

3.4 ZooKeeper Coordination Service

Building a fault-tolerant coordination service for distributed applications is notoriously error prone. An earlier version of CRAQ contained a very simple, centrally-controlled coordination service that maintained membership management. We subsequently opted to leverage ZooKeeper [48], however, to provide CRAQ with a robust, distributed, high-performance method for tracking group membership and an easy way to store chain metadata. Through the use of Zookeeper, CRAQ nodes are guaranteed to receive a notification when nodes are added to or removed from a group. Similarly, a node can be notified when metadata in which it has expressed interest changes.

ZooKeeper provides clients with a hierarchical namespace similar to a filesystem. The filesystem is stored in memory and backed up to a log at each ZooKeeper instance, and the filesystem state is replicated across multiple ZooKeeper nodes for reliability and scalability. To reach agreement, ZooKeeper nodes use an atomic broad-

cast protocol similar to two-phase-commit. Optimized for read-mostly, small-sized workloads, ZooKeeper provides good performance in the face of many readers since it can serve the majority of requests from memory.

Similar to traditional filesystem namespaces, ZooKeeper clients can list the contents of a directory, read the value associated with a file, write a value to a file, and receive a notification when a file or directory is modified or deleted. ZooKeeper’s primitive operations allow clients to implement many higher-level semantics such as group membership, leader election, event notification, locking, and queuing.

Membership management and chain metadata across multiple datacenters does introduce some challenges. In fact, ZooKeeper is not optimized for running in a multi-datacenter environment: Placing multiple ZooKeeper nodes within a single datacenter improves Zookeeper read scalability within that datacenter, but at the cost of wide-area performance. Since the vanilla implementation has no knowledge of datacenter topology or notion of hierarchy, coordination messages between Zookeeper nodes are transmitted over the wide-area network multiple times. Still, our current implementation ensures that CRAQ nodes always receive notifications from *local* Zookeeper nodes, and they are further notified only about chains and node lists that are relevant to them. We expand on our coordination through Zookeeper in §5.1.

To remove the redundancy of cross-datacenter ZooKeeper traffic, one could build a hierarchy of Zookeeper instances: Each datacenter could contain its own local ZooKeeper instance (of multiple nodes), as well as having a representative that participates in the global ZooKeeper instance (perhaps selected through leader election among the local instance). Separate functionality could then coordinate the sharing of data between the two. An alternative design would be to modify ZooKeeper itself to make nodes aware of network topology, as CRAQ currently is. We have yet to fully investigate either approach and leave this to future work.

4 Extensions

This section discusses some additional extensions to CRAQ, including its facility with mini-transactions and the use of multicast to optimize writes. We are currently in the process of implementing these extensions.

4.1 Mini-Transactions on CRAQ

The whole-object read/write interface of an object store may be limiting for some applications. For example, a BitTorrent tracker or other directory service would want to support list addition or deletion. An analytics service

may wish to store counters. Or applications may wish to provide conditional access to certain objects. None of these are easy to provide only armed with a pure object-store interface as described so far, but CRAQ provides key extensions that support transactional operations.

4.1.1 Single-Key Operations

Several single-key operations are trivial to implement, which CRAQ already supports:

- **Prepend/Append:** Adds data to the beginning or end of an object’s current value.
- **Increment/Decrement:** Adds or subtracts to a key’s object, interpreted as an integer value.
- **Test-and-Set:** Only update a key’s object if its current version number equals the version number specified in the operation.

For Prepend/Append and Increment/Decrement operations, the head of the chain storing the key’s object can simply apply the operation to the latest version of the object, even if the latest version is dirty, and then propagate a full replacement write down the chain. Furthermore, if these operations are frequent, the head can buffer the requests and batch the updates. These enhancements would be much more expensive using a traditional two-phase-commit protocol.

For the test-and-set operation, the head of the chain checks if its most recent committed version number equals the version number specified in the operation. If there are no outstanding uncommitted versions of the object, the head accepts the operation and propagates an update down the chain. If there are outstanding writes, we simply reject the test-and-set operation, and clients are careful to back off their request rate if continuously rejected. Alternatively, the head could “lock” the object by disallowing writes until the object is clean and recheck the latest committed version number, but since it is very rare that an uncommitted write is aborted and because locking the object would significantly impact performance, we chose not to implement this alternative.

The test-and-set operation could also be designed to accept a value rather than a version number, but this introduces additional complexity when there are outstanding uncommitted versions. If the head compares against the most recent committed version of the object (by contacting the tail), any writes that are currently in progress would not be accounted for. If instead the head compares against the most recent uncommitted version, this violates consistency guarantees. To achieve consistency, the head would need to temporarily lock the object by disallowing (or temporarily delaying) writes until the object is clean. This does not violate consistency guarantees and ensures

that no updates are lost, but could significantly impact write performance.

4.1.2 Single-Chain Operations

Sinfonia’s recently proposed “mini-transactions” provide an attractive lightweight method [2] of performing transactions on multiple keys within a single chain. A mini-transaction is defined by a compare, read, and write set; Sinfonia exposes a linear address space across many memory nodes. A compare set tests the values of the specified address location and, if they match the provided values, executes the read and write operations. Typically designed for settings with low write contention, Sinfonia’s mini-transactions use an optimistic two-phase commit protocol. The prepare message attempts to grab a lock on each specified memory address (either because different addresses were specified, or the same address space is being implemented on multiple nodes for fault tolerance). If all addresses can be locked, the protocol commits; otherwise, the participant releases all locks and retries later.

CRAQ’s chain topology has some special benefits for supporting similar mini-transactions, as applications can designate multiple objects be stored on the same chain—*i.e.*, those that appear regularly together in multi-object mini-transactions—in such a way that preserves locality. Objects sharing the same *chainid* will be assigned the same node as their chain head, reducing the two-phase commit to a single interaction because only one head node is involved. CRAQ is unique in that mini-transactions that only involve a single chain can be accepted using only the single head to mediate access, as it controls write access to all of a chain’s keys, as opposed to all chain nodes. The only trade-off is that write throughput may be affected if the head needs to wait for keys in the transaction to become clean (as described in §4.1.1). That said, this problem is only worse in Sinfonia as it needs to wait (by exponentially backing off the mini-transaction request) for unlocked keys across multiple nodes. Recovery from failure is similarly easier in CRAQ as well.

4.1.3 Multi-Chain Operations

Even when multiple chains are involved in multi-object updates, the optimistic two-phase protocol need only be implemented with the chain heads, not all involved nodes. The chain heads can lock any keys involved in the mini-transaction until it is fully committed.

Of course, application writers should be careful with the use of extensive locking and mini-transactions: They reduce the write throughput of CRAQ as writes to the same object can no longer be pipelined, one of the very benefits of chain replication.

4.2 Lowering Write Latency with Multicast

CRAQ can take advantage of multicast protocols [41] to improve write performance, especially for large updates or long chains. Since chain membership is stable between node membership changes, a multicast group can be created for each chain. Within a datacenter, this would probably take the form of a network-layer multicast protocol, while application-layer multicast protocols may be better-suited for wide-area chains. No ordering or reliability guarantees are required from these multicast protocols.

Then, instead of propagating a full write serially down a chain, which adds latency proportional to the chain length, the actual value can be multicast to the entire chain. Then, only a small metadata message needs to be propagated down the chain to ensure that all replicas have received a write before the tail. If a node does not receive the multicast for any reason, the node can fetch the object from its predecessor after receiving the write commit message and before further propagating the commit message.

Additionally, when the tail receives a propagated write request, a multicast acknowledgment message can be sent to the multicast group instead of propagating it backwards along the chain. This reduces both the amount of time it takes for a node’s object to re-enter the clean state after a write, as well as the client’s perceived write delay. Again, no ordering or reliability guarantees are required when multicasting acknowledgments—if a node in the chain does not receive an acknowledgement, it will re-enter the clean state when the next read operation requires it to query the tail.

5 Management and Implementation

Our prototype implementation of Chain Replication and CRAQ is written in approximately 3,000 lines of C++ using the Tame extensions [31] to the SFS asynchronous I/O and RPC libraries [38]. All network functionality between CRAQ nodes is exposed via Sun RPC interfaces.

5.1 Integrating ZooKeeper

As described in §3.4, CRAQ needs the functionality of a group membership service. We use a ZooKeeper file structure to maintain node list membership within each datacenter. When a client creates a file in ZooKeeper, it can be marked as *ephemeral*. Ephemeral files are automatically deleted if the client that created the file disconnects from ZooKeeper. During initialization, a CRAQ node creates an ephemeral file in `/nodes/dc_name/node_id`, where `dc_name` is the unique name of its datacenter (as specified by an administrator) and `node_id` is a node identifier unique to the

node’s datacenter. The content of the file contains the node’s IP address and port number.

CRAQ nodes can query `/nodes/dc_name` to determine the membership list for its datacenter, but instead of having to periodically check the list for changes, ZooKeeper provides processes with the ability to create a *watch* on a file. A CRAQ node, after creating an ephemeral file to notify other nodes it has joined the system, creates a watch on the children list of `/nodes/dc_name`, thereby guaranteeing that it receives a notification when a node is added or removed.

When a CRAQ node receives a request to create a new chain, a file is created in `/chains/chain_id`, where `chain_id` is a 160-bit unique identifier for the chain. The chain’s placement strategy (defined in §3.1) determines the contents of the file, but it only includes this chain configuration information, not the list of a chain’s current nodes. Any node participating in the chain will query the chain file and place a watch on it as to be notified if the chain metadata changes.

Although this approach requires that nodes keep track of the CRAQ node list of entire datacenters, we chose this method over the alternative approach in which nodes register their membership for each chain they belong to (*i.e.*, chain metadata explicitly names the chain’s current members). We make the assumption that the number of chains will generally be at least an order of magnitude larger than the number of nodes in the system, or that chain dynamism may be significantly greater than nodes joining or leaving the system (recall that CRAQ is designed for managed datacenter, not peer-to-peer, settings). Deployments where the alternate assumptions hold can take the other approach of tracking per-chain memberships explicitly in the coordination service. If necessary, the current approach’s scalability can also be improved by having each node track only a subset of datacenter nodes: We can partition node lists into separate directories within `/nodes/dc_name/` according to `node_id` prefixes, with nodes monitoring just their own and nearby prefixes.

It is worth noting that we were able to integrate ZooKeeper’s asynchronous API functions into our codebase by building tame-style wrapper functions. This allowed us to *twait* on our ZooKeeper wrapper functions which vastly reduced code complexity.

5.2 Chain Node Functionality

Our `chainnode` program implements most of CRAQ’s functionality. Since much of the functionality of Chain Replication and CRAQ is similar, this program operates as either a Chain Replication node or a CRAQ node based on a run-time configuration setting.

Nodes generate a random identifier when joining the system, and the nodes within each datacenter organize

themselves into a one-hop DHT [29, 45] using these identifiers. A node’s chain predecessor and successor are defined as its predecessor and successor in the DHT ring. Chains are also named by 160-bit identifiers. For a chain C_i , the DHT successor node for C_i is selected as the chain’s first node in that datacenter. In turn, this node’s S DHT successors complete the datacenter subchain, where S is specified in chain metadata. If this datacenter is the chain’s first (resp. last), than this first (resp. last) node is the chain’s ultimate head (resp. tail).

All RPC-based communication between nodes, or between nodes and clients, is currently over TCP connections (with Nagle’s algorithm turned off). Each node maintains a pool of connected TCP connections with its chain’s predecessor, successor, and tail. Requests are pipelined and round-robin’ed across these connections. All objects are currently stored only in memory, although our storage abstraction is well-suited to use an in-process key-value store such as BerkeleyDB [40], which we are in the process of integrating.

For chains that span across multiple datacenters, the last node of one datacenter maintains a connection to the first node of its successor datacenter. Any node that maintains a connection to a node outside of its datacenter must also place a watch on the node list of the external datacenter. Note, though, that when the node list changes in an external datacenter, nodes subscribing to changes will receive notification from their local ZooKeeper instance only, avoiding additional cross-datacenter traffic.

5.3 Handling Memberships Changes

For normal write propagation, CRAQ nodes follow the protocol in §2.3. A second type of propagation, called back-propagation, is sometimes necessary during recovery, however: It helps maintain consistency in response to node additions and failures. For example, if a new node joins CRAQ as the head of an existing chain (given its position in the DHT), the previous head of the chain needs to propagate its state backwards. But the system needs to also be robust to subsequent failures *during* recovery, which can cascade the need for backwards propagation farther down the chain (*e.g.*, if the now-second chain node fails before completing its back-propagation to the now-head). The original Chain Replication paper did not consider such recovery issues, perhaps because it only described a more centrally-controlled and statically-configured version of chain membership, where new nodes are always added to a chain’s tail.

Because of these possible failure conditions, when a new node joins the system, the new node receives propagation messages both from its predecessor and back-propagation from its successor in order to ensure its correctness. A new node refuses client read requests for a

particular object until it reaches agreement with its successor. In both methods of propagation, nodes may use set reconciliation algorithms to ensure that only needed objects are actually propagated during recovery.

Back-propagation messages always contain a node's full state about an object. This means that rather than just sending the latest version, the latest clean version is sent along with all outstanding (newer) dirty versions. This is necessary to enable new nodes just joining the system to respond to future acknowledgment messages. Forward propagation supports both methods. For normal writes propagating down the chain, only the latest version is sent, but when recovering from failure or adding new nodes, full state objects are transmitted.

Let us now consider the following cases from node N 's point of view, where L_C is the length of a chain C for which N is responsible.

Node Additions. A new node, A , is added to the system.

- If A is N 's successor, N propagates all objects in C to A . If A had been in the system before, N can perform object set reconciliation first to identify the specified object versions required to reach consistency with the rest of the chain.
- If A is N 's predecessor:
 - N back-propagates all objects in C to A for which N is not the head.
 - A takes over as the tail of C if N was the previous tail.
 - N becomes the tail of C if N 's successor was previously the tail.
 - A becomes the new head for C if N was previously the head and A 's identifier falls between C and N 's identifier in the DHT.
- If A is within L_C predecessors of N :
 - If N was the tail for C , it relinquishes tail duties and stops participating in the chain. N can now mark its local copies of C 's objects as deletable, although it only recovers this space lazily to support faster state reconciliation if it later rejoins the chain C .
 - If N 's successor was the tail for C , N assumes tail duties.
- If none of the above hold, no action is necessary.

Node Deletions. A node, D , is removed from the system.

- If D was N 's successor, N propagates all objects in C to N 's new successor (again, minimizing transfer to

only unknown, fresh object versions). N has to propagate its objects even if that node already belongs to the chain, as D could have failed before it propagated outstanding writes.

- If D was N 's predecessor:

- N back-propagates all needed objects to N 's new predecessor for which it is not the head. N needs to back-propagate its keys because D could have failed before sending an outstanding acknowledgment to its predecessor, or before finishing its own back-propagation.
- If D was the head for C , N assumes head duties.
- If N was the tail for C , it relinquishes tail duties and propagates all objects in C to N 's new successor.
- If D was within L_C predecessors of N and N was the tail for C , N relinquishes tail duties and propagates all objects in C to N 's new successor.
- If none of the above hold, no action is necessary.

6 Evaluation

This section evaluates the performance of our Chain Replication (CR) and CRAQ implementations. At a high level, we are interested in quantifying the read throughput benefits from CRAQ's ability to apportion reads. On the flip side, version queries still need to be dispatched to the tail for dirty objects, so we are also interested in evaluating asymptotic behavior as the workload mixture changes. We also briefly evaluate CRAQ's optimizations for wide-area deployment.

All evaluations were performed on Emulab, a controlled network testbed. Experiments were run using the *pc3000*-type machines, which have 3GHz processors and 2GB of RAM. Nodes were connected on a 100MBit network. For the following tests, unless otherwise specified, we used a chain size of three nodes storing a single object connected together without any added synthetic latency. This setup seeks to better isolate the performance characteristics of single chains. All graphed data points are the median values unless noted; when present, error bars correspond to the 99th percentile values.

To determine maximal read-only throughput in both systems, we first vary the number of clients in Figure 4, which shows the aggregate read throughput for CR and CRAQ. Since CR has to read from a single node, throughput stays constant. CRAQ is able to read from all three nodes in the chain, so CRAQ throughput increases to three times that of CR. Clients in these experiments maintained

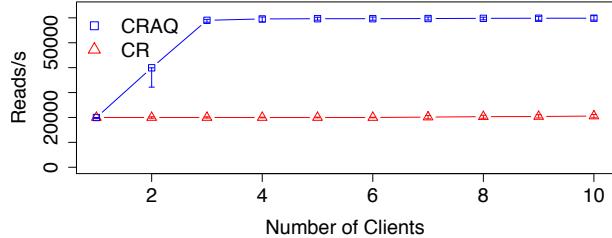


Figure 4: Read throughput as the number of readers increase: A small number of clients can saturate both CRAQ and CR, although CRAQ’s asymptotic behavior scales with chain size, while CR is constant.

Throughput (in operations/s)				
	Type	1st	Median	99th
Read	CR-3	19,590	20,552	21,390
	CRAQ-3	58,998	59,882	60,626
	CRAQ-5	98,919	99,466	100,042
	CRAQ-7	137,390	138,833	139,537
Write	CRAQ-3	5,480	5,514	5,544
	CRAQ-5	4,880	4,999	5,050
	CRAQ-7	4,420	4,538	4,619
Test & Set	CRAQ-3	732	776	877
	CRAQ-5	411	427	495
	CRAQ-7	290	308	341

Figure 5: Throughput of read and write operations for a 500-byte object and throughput for a test-and-set operation incrementing a 4-byte integer.

a maximum window of outstanding requests (50), so the system never entered a potential livelock scenario.

Figure 5 shows throughput for read, write, and test-and-set operations. Here, we varied CRAQ chains from three to seven nodes, while maintaining read-only, write-only, and transaction-only workloads. We see that read throughput scaled linearly with the number of chain nodes as expected. Write throughput decreased as chain length increased, but only slightly. Only one test-and-set operation can be outstanding at a time, so throughput is much lower than for writes. Test-and-set throughput also decreases as chain length increases because the latency for a single operation increases with chain length.

To see how CRAQ performs during a mixed read/write workload, we set ten clients to continuously read a 500-byte object from the chain while a single client varied its write rate to the same object. Figure 6 shows the aggregate read throughput as a function of write rate. Note that

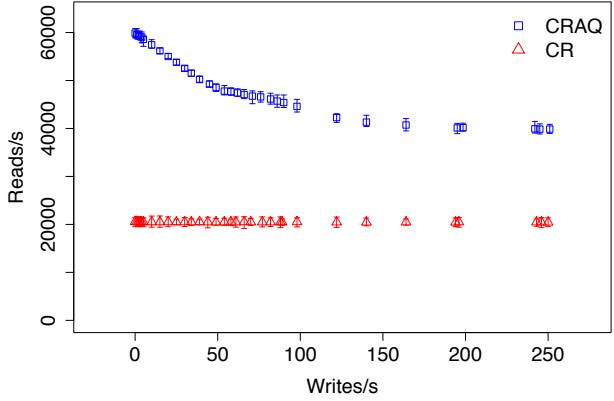


Figure 6: Read throughput on a length-3 chain as the write rate increases (500B object).

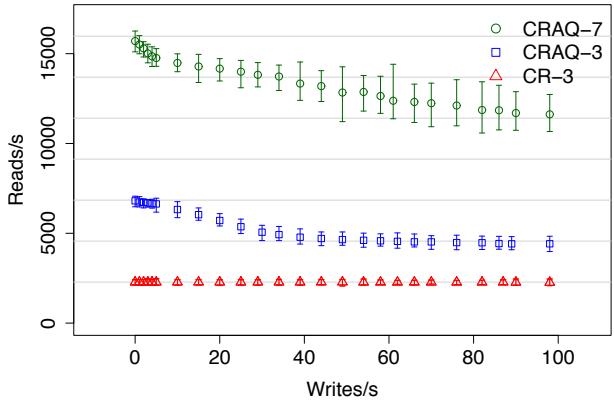


Figure 7: Read throughput as writes increase (5KB object).

Chain Replication is not effected by writes, as all read requests are handled by the tail. Although throughput for CRAQ starts out at approximately three times the rate of CR (a median of 59,882 reads/s vs. 20,552 reads/s), as expected, this rate gradually decreases and flattens out to around twice the rate (39,873 reads/s vs. 20,430 reads/s). As writes saturate the chain, non-tail nodes are always dirty, requiring them always to first perform version requests to the tail. CRAQ still enjoys a performance benefit when this happens, however, as the tail’s saturation point for its combined read and version requests is still higher than that for read requests alone.

Figure 7 repeats the same experiment, but using a 5 KB object instead of a 500 byte one. This value was chosen as a common size for objects such as small Web images, while 500 bytes might be better suited for smaller database entries (*e.g.*, blog comments, social-network status information, etc.). Again, CRAQ’s performance in read-only settings significantly outperforms that of CR with a chain size of three (6,808 vs. 2,275 reads/s), while it preserves good behavior even under high write rates (4,416 vs. 2,259 reads/s). This graph also includes CRAQ performance with seven-node chains. In both scenarios,

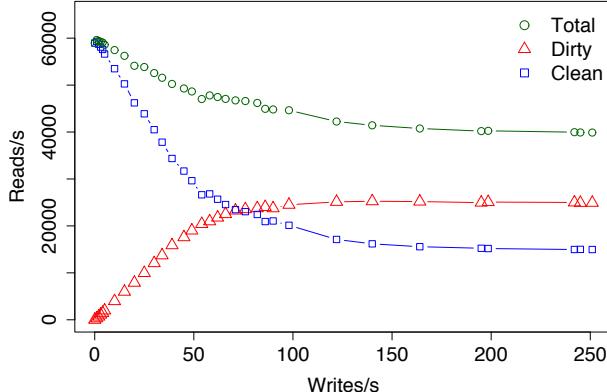


Figure 8: Number of reads that are dirty vs. clean reads as writes increase (500B key).

even as the tail becomes saturated with requests, its ability to answer small version queries at a much higher rate than sending larger read replies allows aggregate read throughput to remain significantly higher than in CR.

Figure 8 isolates the mix of dirty and clean reads that comprise Figure 6. As writes increase, the number of clean requests drops to 25.4% of its original value, since only the tail is clean as writes saturate the chain. The tail cannot maintain its own maximal read-only throughput (*i.e.*, 33.3% of the total), as it now also handles version queries from other chain nodes. On the other hand, the number of dirty requests would approach two-thirds of the original clean read rate if total throughput remained constant, but since dirty requests are slower, the number of dirty requests flattens out at 42.3%. These two rates reconstruct the total observed read rate, which converges to 67.7% of read-only throughput during high write contention on the chain.

The table in Figure 9 shows the latency in milliseconds of clean reads, dirty reads, writes to a 3-node chain, and writes to a 6-node chain, all within a single datacenter. Latencies are shown for objects of 500 bytes and 5 KB both when the operation is the only outstanding request (No Load) and when we saturate the CRAQ nodes with many requests (High Load). As expected, latencies are higher under heavy load, and latencies increase with key size. Dirty reads are always slower than clean reads because of the extra round-trip-time incurred, and write latency increases roughly linearly with chain size.

Figure 10 demonstrates CRAQ’s ability to recover from failure. We show the loss in read-only throughput over time for chains of lengths 3, 5, and 7. Fifteen seconds into each test, one of the nodes in the chain was killed. After a few seconds, the time it takes for the node to time out and be considered dead by ZooKeeper, a new node joins the chain and throughput resumes to its original value. The horizontal lines drawn on the graph correspond to the

Latency (in ms)					
		Type	Size	Med	95th
No Load	Writes	Clean	500	0.49	0.74
		Dirty	500	0.98	0.99
Length 3	Writes	5KB	0.99	1.00	1.23
		5KB	1.24	1.49	1.73
Length 6	Writes	500	2.05	2.29	2.43
		5KB	4.78	5.00	5.05
Clean	Writes	500	4.51	4.93	5.01
		5KB	9.09	9.79	10.05
Heavy Load	Writes	Clean	500	1.49	2.74
		Dirty	500	2.98	5.48
Length 3	Writes	5KB	1.99	3.73	4.22
		5KB	3.50	6.23	7.23
Length 6	Writes	500	5.75	7.26	7.88
		5KB	11.61	14.45	15.72
		500	20.65	21.66	22.09
		5KB	33.72	42.88	43.61

Figure 9: CRAQ Latency by load, chain length, object state, and object size within a single datacenter.

maximum throughput for chains of lengths 1 through 7. This helps illustrate that the loss in throughput during the failure is roughly equal to $1/C$, where C is the length of the chain.

To measure the effect of failure on the latency of read and write operations, Figures 11 and 12 show the latency of these operations during the failure of a chain of length three. Clients that receive an error when trying to read an object choose a new random replica to read from, so failures have a low impact on reads. Writes, however, cannot be committed during the period between when a replica fails and when it is removed from the chain due to timeouts. This causes write latency to increase to the time it takes to complete failure detection. We note that this is the same situation as in any other primary/backup replication strategy which requires all live replicas to participate in commits. Additionally, clients can optionally configure a write request to return as soon as the head of the chain accepts and propagates the request down to the chain instead of waiting for it to commit. This reduces latency for clients that don’t require strong consistency.

Finally, Figure 13 demonstrates CRAQ’s utility in wide-area deployments across datacenters. In this experiment, a chain was constructed over three nodes that each have 80ms of round-trip latency to one another (approximately the round-trip-time between U.S. coastal areas), as controlled using Emulab’s synthetic delay. The read client was not local to the chain tail (which otherwise could have just resulted in local-area performance as before).

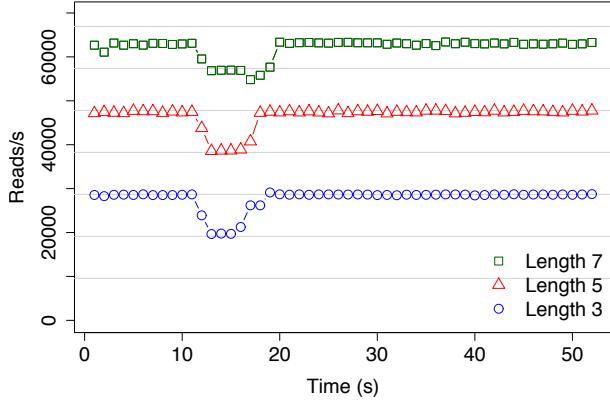


Figure 10: CRAQ re-establishing normal read throughput after a single node in a chain serving a 500-byte object fails.

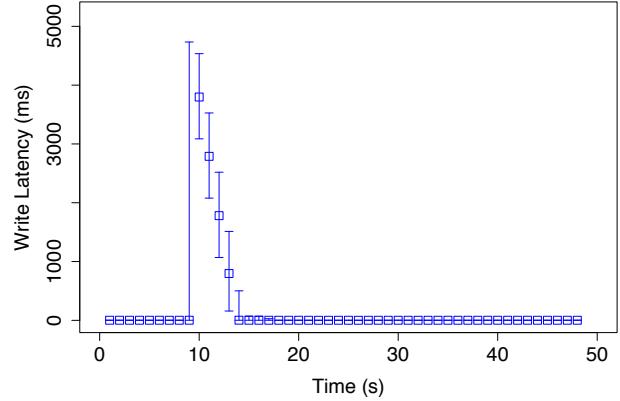


Figure 12: CRAQ's write latency increases during failure, since the chain cannot commit write operations.

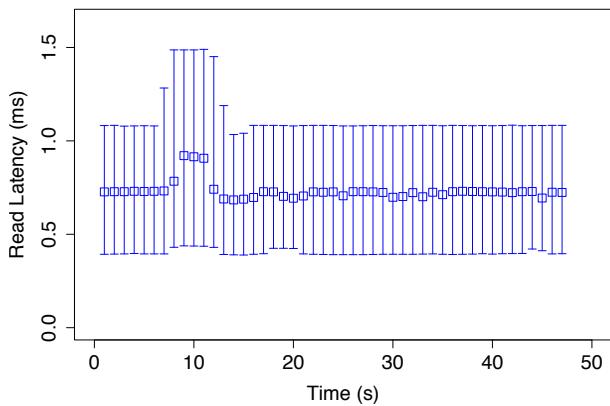


Figure 11: CRAQ's read latency (shown here under moderate load) goes up slightly during failure, as requests to the failed node need to be retried at a non-faulty node.

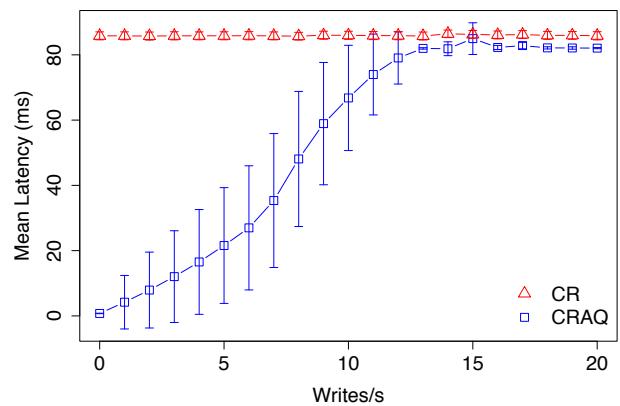


Figure 13: CR and CRAQ's read latency to a local client when the tail is in a distant datacenter separated by an RTT of 80ms and the write rate of a 500-byte object is varied.

The figure evaluates read latency as the workload mixture changes; mean latency is now shown with standard deviation as error bars (as opposed to median and 99th percentile elsewhere). Since the tail is not local, CR's latency remains constantly high, as it always incurs a wide-area read request. CRAQ, on the other hand, incurs almost no latency when no writes are occurring, as the read request can be satisfied locally. As the write rate increases, however, CRAQ reads are increasingly dirty, so the average latency rises. Once the write rate reaches about 15 writes/s, the latency involved in propagating write messages down the wide-area chain causes the client's local node to be dirty 100% of the time, leading to a wide-area version query. (CRAQ's maximum latency is ever-so-slightly less than CR given that only metadata is transferred over the wide area, a difference that would only increase with larger objects, especially in slow-start scenarios.) Although this convergence to a 100% dirty state occurs at a much lower write rate than before, we note that careful chain placement allows any clients in the tail's datacenter to enjoy local-area performance. Further, clients

in non-tail datacenters that can be satisfied with a degree of maximum-bounded inconsistency (per §2.4) can also avoid wide-area requests.

7 Related Work

Strong consistency in distributed systems. Strong consistency among distributed servers can be provided through the use of primary/backup storage [3] and two-phase commit protocols [43]. Early work in this area did not provide for availability in the face of failures (*e.g.*, of the transaction manager), which led to the introduction of view change protocols (*e.g.*, through leader consensus [33]) to assist with recovery. There has been a large body of subsequent work in this area; recent examples include both Chain Replication and the ring-based protocol of Guerraoui *et al.* [25], which uses a two-phase write protocol and delays reads during uncommitted writes. Rather than replicate content everywhere, one can explore other trade-offs between overlapping read and write sets

in strongly-consistent quorum systems [23, 28]. Agreement protocols have also been extended to malicious settings, both for state machine replication [10, 34] and quorum systems [1, 37]. These protocols provide linearizability across *all* operations to the system. This paper does not consider Byzantine faults—and largely restricts its consideration of operations affecting single objects—although it is interesting future work to extend chain replication to malicious settings.

There have been many examples of distributed filesystems that provide strong consistency guarantees, such as the early primary/backup-based Harp filesystem [35]. More recently, Boxwood [36] explores exporting various higher-layer data abstractions, such as a B-tree, while offering strict consistency. Sinfonia [2] provides lightweight “mini-transactions” to allow for atomic updates to exposed memory regions in storage nodes, an optimized two-phase commit protocol well-suited for settings with low write contention. CRAQ’s use of optimistic locking for multi-chain multi-object updates was heavily influenced by Sinfonia.

CRAQ and Chain Replication [47] are both examples of object-based storage systems that expose whole-object writes (updates) and expose a flat object namespace. This interface is similar to that provided by key-value databases [40], treating each object as a row in these databases. As such, CRAQ and Chain Replication focus on strong consistency in the ordering of operations *to each object*, but does not generally describe ordering of operations to different objects. (Our extensions in §4.1 for multi-object updates are an obvious exception.) As such, they can be viewed in light of causal consistency taken to the extreme, where only operations to the same object are causally related. Causal consistency was studied both for optimistic concurrency control in databases [7] and for ordered messaging layers for distributed systems [8]. Yahoo!’s new data hosting service, PNUTS [12], also provides per-object write serialization (which they call per-record timeline consistency). Within a single datacenter, they achieve consistency through a messaging service with totally-ordered delivery; to provide consistency across datacenters, all updates are sent to a local record master, who then delivers updates in committed order to replicas in other datacenters.

The chain self-organization techniques we use are based on those developed by the DHT community [29, 45]. Focusing on peer-to-peer settings, CFS provides a read-only filesystem on top of a DHT [14]; Carbonite explores how to improve reliability while minimizing replica maintenance under transient failures [11]. Strongly-consistent mutable data is considered by OceanStore [32] (using BFT replication at core nodes) and Etna [39] (using Paxos to partition the DHT into smaller replica groups and quorum protocols for con-

sistency). CRAQ’s wide-area solution is more datacenter-focused and hence topology-aware than these systems. Coral [20] and Canon [21] both considered hierarchical DHT designs.

Weakening Consistency for Availability. TACT [49] considers the trade-off between consistency and availability, arguing that weaker consistency can be supported when system constraints are not as tight. eBay uses a similar approach: messaging and storage are eventually-consistent while an auction is still far from over, but use strong consistency—even at the cost of availability—right before an auction closes [46].

A number of filesystems and object stores have traded consistency for scalability or operation under partitions. The Google File System (GFS) [22] is a cluster-based object store, similar in setting to CRAQ. However, GFS sacrifices strong consistency: concurrent writes in GFS are not serialized and read operations are not synchronized with writes. Filesystems designed with weaker consistency semantics include Sprite [6], Coda [30], Ficus [27], and Bayou [42], the latter using epidemic protocols to perform data reconciliation. A similar gossip-style anti-entropy protocol is used in Amazon’s Dynamo object service [15], to support “always-on” writes and continued operation when partitioned. Facebook’s new Cassandra storage system [16] also offers only eventual consistency. The common use of memcached [18] with a relational database does not offer any consistency guarantees and instead relies on correct programmer practice; maintaining even loose cache coherence across multiple datacenters has been problematic [44].

CRAQ’s strong consistency protocols do not support writes under partitioned operation, although partitioned chain segments can fall back to read-only operation. This trade-off between consistency, availability, and partition-tolerance was considered by BASE [19] and Brewer’s CAP conjecture [9].

8 Conclusions

This paper presented the design and implementation of CRAQ, a successor to the chain replication approach for strong consistency. CRAQ focuses on scaling out read throughput for object storage, especially for read-mostly workloads. It does so by supporting *apportioned queries*: that is, dividing read operations over all nodes of a chain, as opposed to requiring that they all be handled by a single primary node. While seemingly simple, CRAQ demonstrates performance results with significant scalability improvements: proportional to the chain length with little write contention—*i.e.*, 200% higher throughput with three-node chains, 600% with seven-node chains—and,

somewhat surprisingly, still noteworthy throughput improvements when object updates are common.

Beyond this basic approach to improving chain replication, this paper focuses on realistic settings and requirements for a chain replication substrate to be useful across a variety of higher-level applications. Along with our continued development of CRAQ for multi-site deployments and multi-object updates, we are working to integrate CRAQ into several other systems we are building that require reliable object storage. These include a DNS service supporting dynamic service migration, rendezvous servers for a peer-assisted CDN [5], and a large-scale virtual world environment. It remains as interesting future work to explore these applications' facilities in using both CRAQ's basic object storage, wide-area optimizations, and higher-level primitives for single-key and multi-object updates.

Acknowledgments

The authors would like to thank Wyatt Lloyd, Muneeb Ali, Siddhartha Sen, and our shepherd Alec Wolman for helpful comments on earlier drafts of this paper. We also thank the Flux Research Group at Utah for providing access to the Emulab testbed. This work was partially funded under NSF NeTS-ANET Grant #0831374.

References

- [1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.
- [2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.
- [3] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proc. Intl. Conference on Software Engineering*, Oct. 1976.
- [4] Amazon. S3 Service Level Agreement. <http://aws.amazon.com/s3-sla/>, 2009.
- [5] C. Aperjis, M. J. Freedman, and R. Johari. Peer-assisted content distribution with prices. In *Proc. SIGCOMM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, Dec. 2008.
- [6] M. Baker and J. Ousterhout. Availability in the Sprite distributed file system. *Operating Systems Review*, 25(2), Apr. 1991.
- [7] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proc. Very Large Data Bases (VLDB)*, Oct. 1980.
- [8] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), 1993.
- [9] E. Brewer. Towards robust distributed systems. *Principles of Distributed Computing (PODC) Keynote*, July 2000.
- [10] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. Operating Systems Design and Implementation (OSDI)*, Feb. 1999.
- [11] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. Networked Systems Design and Implementation (NSDI)*, May 2006.
- [12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proc. Very Large Data Bases (VLDB)*, Aug. 2008.
- [13] CouchDB. <http://couchdb.apache.org/>, 2009.
- [14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.
- [16] Facebook. Cassandra: A structured storage system on a P2P network. <http://code.google.com/p/the-cassandra-project/>, 2009.
- [17] Facebook. Infrastructure team. Personal Comm., 2008.
- [18] B. Fitzpatrick. Memcached: a distributed memory object caching system. <http://www.danga.com/memcached/>, 2009.
- [19] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.
- [20] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. Networked Systems Design and Implementation (NSDI)*, Mar. 2004.
- [21] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in G Major: Designing DHTs with hierarchical structure. In *Proc. Intl. Conference on Distributed Computing Systems (ICDCS)*, Mar. 2004.
- [22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.
- [23] D. K. Gifford. Weighted voting for replicated data. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Dec. 1979.
- [24] Google. Google Apps Service Level Agreement. <http://www.google.com/apps/intl/en/terms/sla.html>, 2009.
- [25] R. Guerraoui, D. Kostic, R. R. Levy, and V. Quéma. A high throughput atomic storage algorithm. In *Proc. Intl. Conference on Distributed Computing Systems (ICDCS)*, June 2007.

- [26] D. Hakala. Top 8 datacenter disasters of 2007. *IT Management*, Jan. 28 2008.
- [27] J. Heidemann and G. Popek. File system development with stackable layers. *ACM Trans. Computer Systems*, 12(1), Feb. 1994.
- [28] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Computer Systems*, 4(1), Feb. 1986.
- [29] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. Symposium on the Theory of Computing (STOC)*, May 1997.
- [30] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Computer Systems*, 10(3), Feb. 1992.
- [31] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proc. USENIX Annual Technical Conference*, June 2007.
- [32] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherpoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov 2000.
- [33] L. Lamport. The part-time parliament. *ACM Trans. Computer Systems*, 16(2), 1998.
- [34] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Programming Language Systems*, 4(3), 1982.
- [35] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Aug. 1991.
- [36] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. Operating Systems Design and Implementation (OSDI)*, Dec. 2004.
- [37] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. Symposium on the Theory of Computing (STOC)*, May 1997.
- [38] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Dec 1999.
- [39] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: a fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT, June 2005.
- [40] Oracle. BerkeleyDB v4.7, 2009.
- [41] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. RFC 1546, Network Working Group, Nov. 1993.
- [42] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, , and A. Demers. Flexible update propagation for weakly consistent replication. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.
- [43] D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Trans. Software Engineering*, 9(3), May 1983.
- [44] J. Sobel. Scaling out. Engineering at Facebook blog, Aug. 20 2008.
- [45] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Networking*, 11, 2002.
- [46] F. Travostino and R. Shoup. eBay’s scalability odyssey: Growing and evolving a large ecommerce site. In *Proc. Large-Scale Distributed Systems and Middleware (LADIS)*, Sept. 2008.
- [47] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. Operating Systems Design and Implementation (OSDI)*, Dec. 2004.
- [48] Yahoo! Hadoop Team. Zookeeper. <http://hadoop.apache.org/zookeeper/>, 2009.
- [49] H. Yu and A. Vahdat. The cost and limits of availability for replicated services. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

Census: Location-Aware Membership Management for Large-Scale Distributed Systems

James Cowling^{*} Dan R. K. Ports^{*} Barbara Liskov^{*}
Raluca Ada Popa^{*} Abhijeet Gaikwad[†]
MIT CSAIL^{}* *École Centrale Paris[†]*

Abstract

We present *Census*, a platform for building large-scale distributed applications. *Census* provides a membership service and a multicast mechanism. The membership service provides every node with a consistent view of the system membership, which may be global or partitioned into location-based regions. *Census* distributes membership updates with low overhead, propagates changes promptly, and is resilient to both crashes and Byzantine failures. We believe that *Census* is the first system to provide a consistent membership abstraction at very large scale, greatly simplifying the design of applications built atop large deployments such as multi-site data centers.

Census builds on a novel multicast mechanism that is closely integrated with the membership service. It organizes nodes into a reliable overlay composed of multiple distribution trees, using network coordinates to minimize latency. Unlike other multicast systems, it avoids the cost of using distributed algorithms to construct and maintain trees. Instead, each node independently produces the same trees from the consistent membership view. *Census* uses this multicast mechanism to distribute membership updates, along with application-provided messages.

We evaluate the platform under simulation and on a real-world deployment on PlanetLab. We find that it imposes minimal bandwidth overhead, is able to react quickly to node failures and changes in the system membership, and can scale to substantial size.

1 Introduction

Today's increasingly large-scale distributed systems must adapt to dynamic membership, providing efficient and reliable service despite churn and failures. Such systems typically incorporate or rely on some sort of *membership service*, which provides the application with information about the nodes in the system. The current shift toward cloud computing and large multi-site data centers provides further motivation for a system designed to manage node membership at this large scale.

Many membership services exist, with varying semantics. Some, such as those based on virtual synchrony, provide strict semantics, ensuring that each node sees a consistent view of the system membership, but operate only at small scales [3, 14, 37]. Because maintaining consistency and global membership knowledge is often perceived as prohibitively expensive, many recently proposed systems provide weaker semantics. These systems provide greater scalability, but make no guarantees about members having consistent views [16, 17], and some provide only partial views of system membership [36, 23, 32].

We argue that it is both feasible and useful to maintain consistent views of system membership even in large-scale distributed systems. We present *Census*, a new platform for constructing such applications, consisting of a membership management system and a novel multicast mechanism. The membership management system follows the virtual synchrony paradigm: the system divides time into epochs, and all nodes in the same epoch have identical views of system membership. This globally consistent membership view represents a powerful abstraction that simplifies the design of applications built atop our platform. In addition to eliminating the need for applications to build their own system for detecting and tracking membership changes, globally consistent views can simplify application protocol design.

Census is designed to work at large scale, even with highly-dynamic membership, and to tolerate both crashes and Byzantine failures. It uses three main techniques to achieve these goals.

First, *Census* uses a locality-based hierarchical organization. Nodes are grouped into *regions* according to their network coordinates. Even in small systems, this hierarchical structure is used to reduce the costs of aggregating reports of membership changes. For systems so large that it is infeasible to maintain a global membership view, we provide a *partial knowledge* deployment option, where nodes know the full membership of their own region but only a few representative nodes from each other region.

Second, Census uses a novel multicast mechanism that is closely intertwined with the membership management service. The membership service relies on the multicast mechanism to distribute update notifications, and the multicast system constructs its distribution trees using node and location information from the membership service. The overlay topology, made up of redundant interior-disjoint trees, is similar to other systems [7, 40]. However, the trees are constructed in a very different way: each node independently carries out a deterministic tree construction algorithm when it receives a membership update. This eliminates the need for complex and potentially expensive distributed tree-building protocols, yet it produces efficient tree structures and allows the trees to change frequently to improve fault-tolerance. We also take advantage of global membership and location information to keep bandwidth overhead to a minimum by ensuring that each node receives no redundant data, while keeping latency low even if there are failures.

Finally, Census provides fault-tolerance. Unlike systems that require running an agreement protocol among all nodes in the system [30, 20], Census uses only a small subset of randomly-chosen nodes, greatly reducing the costs of membership management while still providing correctness with high probability. In most cases, we use lightweight quorum protocols to avoid the overhead of full state machine replication. We also discuss several new issues that arise in a Byzantine environment.

Census exposes the region abstraction and multicast mechanism to applications as additional services. Regions can be a useful organizing technique for applications. For example, a cooperative caching system might use regions to determine which nodes share their caches. The multicast system provides essential functionality for many applications that require membership knowledge, since a membership change may trigger a system reconfiguration (*e.g.* changing responsible nodes in a distributed storage system) that must be announced to all nodes.

Our evaluation of Census, under simulation and in a real-world deployment on PlanetLab, indicates that it imposes low bandwidth overhead per node (typically less than 1 KB/s even in very large systems), reacts quickly to node failures and system membership changes, and can scale to substantial size (over 100,000 nodes even in a high-churn environment).

The remainder of this paper is organized as follows. We define our assumptions in Section 2. Sections 3–5 describe Census’s architecture, multicast mechanism, and fault-tolerance strategy in detail. Section 6 presents performance results based on both theoretical analysis and a deployment on PlanetLab. We sketch some ways applications can use the platform in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 Model and Assumptions

Census is intended to be used in an asynchronous network like the Internet, in which messages may be corrupted, lost or reordered. We assume that messages sent repeatedly will eventually be delivered. We also assume nodes have loosely synchronized clock rates, such that they can approximately detect the receipt of messages at regular intervals. Loosely synchronized clock rates are easy to guarantee in practice, unlike loosely synchronized clocks.

Every node in our platform has an IP address, a unique random ID, and network coordinates. Tolerating Byzantine failures adds a few more requirements. Each node must have a public key, and its unique ID is assigned by taking a collision-resistant hash of the public key. Furthermore, we require admission control to prevent Sybil attacks [13], so each joining node must present a certificate signed by a trusted authority vouching for its identity.

All nodes have coordinates provided by a network coordinate system such as Vivaldi [11]. We describe the system in terms of a two-dimensional coordinate system plus *height*, analogous to the last-hop network delay. This follows the model used in Vivaldi, but our system could easily be modified to use a different coordinate space. We assume coordinates reflect network latency, but their accuracy affects only performance, not correctness.

Traditional network coordinate systems do not function well in a Byzantine environment since malicious nodes can influence the coordinates of honest nodes [38]. We have developed a protocol [39] that ensures that honest nodes’ coordinates accurately reflect their locations by using a group of landmark nodes, some of which are permitted to be faulty. Another approach is described in [34]. These techniques do not provide any guarantees about the accuracy of a Byzantine node’s coordinates, and we do not assume any such guarantees.

3 Platform Architecture

Our system moves through a sequence of *epochs*, numbered sequentially. Each epoch has a particular membership view. One of the members acts as the *leader*. Nodes inform the leader of membership events (nodes joining or leaving) and the leader collects this information for the duration of the epoch. The epoch length is a parameter whose setting depends on application needs and assumptions about the environment; for example, our experiments use 30s epochs. Users may opt to place the leader on a fixed node, or select a new leader each epoch based on the system membership and epoch number.

At the end of an epoch, the leader creates an *item* containing the membership changes and next epoch number, and multicasts this information as described in Section 4. The item can also include data provided by the application. The leader makes an upcall to the application code at its node to obtain this data and includes it in the item.

In addition, the system can perform additional multicasts within an epoch to propagate application data if desired.

When a node receives an item, it updates its view of the membership to reflect the latest joins and departures, then *enters* the next epoch. It can only process the item if it knows the system state of the previous epoch; nodes keep a few recent items in a log to enable nodes that are slightly behind to obtain missing information.

Our system ensures *consistency*: all nodes in the same epoch have identical views of the membership. The multicast mechanism delivers items quickly and reliably, so that nodes are likely to be in the same epoch at the same time. Messages include the epoch number at the point they were sent, to ensure they are routed and processed with respect to the correct membership view. Applications that require consistency also include the current epoch number in application messages, only processing messages when the sender and receiver agree on the epoch.

In this section, we describe how the system is organized. We begin in Section 3.1 with a simplified version with only a simple region. In Section 3.2, we introduce the multi-region structure, which improves scalability even though all nodes still know the membership of the entire system. Finally, in Section 3.3, we describe an optional extension to the system for extremely large or dynamic environments, where each node has full membership information only for its own region.

3.1 Single-Region Deployment

In a one-region system, all membership events are processed directly by the leader. The leader gathers notifications of node joins and departures throughout the epoch, then aggregates them into an item and multicasts the item to the rest of the system, starting the next epoch.

To join the system, a node sends a message identifying itself to the leader, providing its network coordinates and identity certificate (if tolerating Byzantine faults). The leader verifies the certificate, adds the node to a list of new joiners, and informs the new node of the epoch number and a few current members. The new node obtains the current membership from one of these nodes, reducing the load on the leader.

To remove a node, a departure request is sent to the leader identifying the node to be removed. A node can leave the system gracefully by requesting its own removal (in a Byzantine environment, this request must be signed). Nodes that do not fail gracefully are reported by other nodes; Section 5 describes this process. If the request is valid, the leader adds the node to a list of departures.

Nodes include their coordinates in the join request, ensuring that all nodes see a consistent view of each other's coordinates. Node locations can change over time, however, and coordinates should continue to reflect network proximity. Each node monitors its coordinates and reports

changes, which are propagated in the next item. To avoid instability, nodes report only major location changes, using a threshold.

3.2 Multi-Region Deployment

Even at relatively high churn, with low available bandwidth and CPU resources, our analysis indicates that the single-region structure scales to beyond 10,000 nodes. As the system grows, however, the request load on the leader, and the overhead in computing distribution trees, increases. To accommodate larger systems, we provide a structure in which the membership is divided into regions based on proximity. Each region has a region ID and every node belongs to exactly one region. Even in relatively small systems, the multi-region structure is useful to reduce load on the leader, and to provide the application with locality-based regions.

In a multi-region system each region has its own local leader, which can change each epoch. This *region leader* collects joins and departures for nodes in its region. Towards the end of the epoch, it sends a *report* listing these membership events to the global leader, and the leader propagates this information in the next item. Any membership events that are received too late to be included in the report are forwarded to the next epoch's leader.

Even though all nodes still know the entire system membership, this architecture is more scalable. It offloads work from the global leader in two ways. First, the leader processes fewer messages, since it only handles aggregate information about joins and departures. Second, it can offload some cryptographic verification tasks, such as checking a joining node's certificate, to the region leaders. Moreover, using regions also reduces the CPU costs of our multicast algorithm, as Section 4 describes: nodes need not compute full distribution trees for other regions.

To join the system, a node contacts any member of the system (discovered out-of-band) and sends its coordinates. The member redirects the joining node to the leader of the region whose centroid is closest to the joining node. When a node's location changes, it may find that a different region is a better fit for it. When this happens, the node uses a *move* request to inform the new region's leader that it is leaving another region. This request is sent to the global leader and propagated in the next item.

3.2.1 Region Dynamics: Splitting and Merging

Initially, the system has only one region. New regions are formed by splitting existing regions when they grow too large. Similarly, regions that grow too small can be removed by merging them into other regions.

The global leader tracks the sizes of regions and when one of them exceeds a *split threshold*, it tells that region to split by including a *split request* in the next item. This request identifies the region that should split, and provides

the ID to be used for the newly formed region. When a region’s size falls below a *merge threshold*, the leader selects a neighboring region for it to merge into, and inserts a *merge request* containing the two region IDs in the next item. The merge threshold is substantially smaller than the split threshold, to avoid oscillation.

Whenever a node processes an item containing a split or merge request, it carries out the split or merge computation. For a split, it computes the centroid and the widest axis, then splits the region into two parts. The part to the north or west retains the region ID, and the other part is assigned the new ID. For a merge, nodes from one region are added to the membership of the second region. As soon as this item is received, nodes consider themselves members of their new region.

3.3 Partial Knowledge

Even with the multi-region structure, scalability is ultimately limited by the need for every membership event in an epoch to be broadcast in the next item. The bandwidth costs of doing so are proportional to the number of nodes and the churn rate. For most systems, this cost is reasonable; our analysis in Section 6.1 shows, for example, that for systems with 100,000 nodes, even with a very short average node lifetime (30 minutes), average bandwidth overhead remains under 5 KB/s. However, for extremely large, dynamic, and/or bandwidth-constrained environments, the updates may grow too large.

For such systems, we provide a *partial knowledge* deployment option. Here, nodes have complete knowledge of the members of their own region, but know only *summary* information about other regions. We still provide consistency, however: in a particular epoch, every node in a region has the same view of the region, and every node in the system has the same view of all region summaries.

In this system, region leaders send the global leader only a summary of the membership changes in the last epoch, rather than the full report of all joins and departures. The summary identifies the region leader for the next epoch, provides the size and centroid of the region, and identifies some region members that act as its global *representatives*. The global leader includes this message in the next item, propagating it to all nodes in the system.

As we will discuss in Section 4, the representatives are used to build distribution trees. In addition, the representatives take care of propagating the full report, containing the joins and leaves, to nodes in their region; this way nodes in the region can compute the region membership. The region leader sends the report to the representatives at the same time it sends the summary to the global leader.

3.3.1 Splitting and Merging with Partial Knowledge

Splits and merges are operations involving the membership of multiple regions, so they are more complex in a

partial knowledge deployment where nodes do not know the membership of other regions. We extend the protocols to transfer the necessary membership information between the regions involved.

When a region s is merged into neighboring region t , members of both regions need to learn the membership of the other. The representatives of s and t communicate to exchange this information, then propagate it on the tree for their region. The leader for t sends the global leader a summary for the combined region, and nodes in s consider themselves members of t as soon as they receive the item containing this summary.

A split cannot take place immediately because nodes outside the region need to know the summary information (centroid, representatives, etc.) for the newly-formed regions and cannot compute it themselves. When the region’s leader receives a split request, it processes joins and leaves normally for the remainder of the epoch. At the end of the epoch, it carries out the split computation, and produces two summaries, one for each new region. These summaries are distributed in the next item, and the split takes effect in the next epoch.

4 Multicast

This section describes our multicast mechanism, which is used to disseminate membership updates and application data. The goals of the design are ensuring reliable delivery despite node failures and minimizing bandwidth overhead. Achieving low latency and a fair distribution of forwarding load are also design considerations.

Census’s multicast mechanism uses multiple distribution trees, like many other multicast systems. However, our trees are constructed in a different way, taking advantage of the fact that membership information is available at all nodes. Trees are constructed on-the-fly using a deterministic algorithm on the system membership: as soon as a node receives the membership information for an epoch from one of its parents, it can construct the distribution tree, and thereby determine which nodes are its children. Because the algorithm is deterministic, each node computes exactly the same trees.

This use of global membership state stands in contrast to most multicast systems, which instead try to minimize the amount of state kept by each node. Having global membership information allows us to run what is essentially a centralized tree construction algorithm at each node, rather than a more complex distributed algorithm.

Our trees are constructed anew each epoch, ignoring their structure from the previous epoch. This may seem surprising, in light of the conventional wisdom that “stability of the routing trees is very important to achieve workable, reliable routing” [2]. However, this statement applies to multicast protocols that require executing costly protocols to change the tree structure, and may experi-

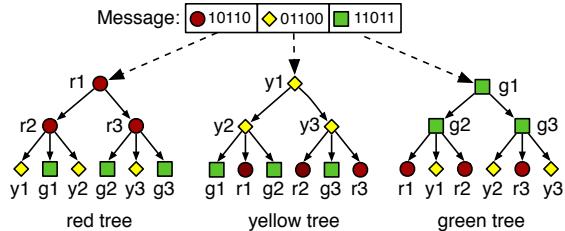


Figure 1: Distribution trees for a 3-color deployment. All nodes are members of each tree, but an internal node is only one. For example, the red nodes r_1-r_3 are interior nodes in the red tree, and leaves in the other two.

ence oscillatory or transitory behavior during significant adjustments. Our approach allows the trees to be recomputed with no costs other than those of maintaining the membership information. Changing the trees can also improve fault-tolerance and load distribution because different nodes are located at or near the root of the tree [24].

4.1 Multiple-Tree Overlay

A multicast overlay consisting of a single tree is insufficient to ensure reliability and fair load distribution: the small group of interior nodes bears all the forwarding load while the leaves bear none, and an interior node failure leads to loss of data at all its descendants. Instead, Census uses multiple trees, as in SplitStream [7], to spread forwarding load more evenly and enhance reliability.

Our overlay consists of a set of trees, typically between 4 and 16. The trees are *interior-node-disjoint*: each node is an interior node in at most one tree. We refer to each tree by a *color*, with each node in the system also assigned a color. The interior of the *red* tree is composed completely of red nodes, ensuring our disjointness constraint. The nodes of other colors form the leaves of the red tree, and the red nodes are leaf nodes in all other trees. Using an even distribution of node colors and a fan-out equal to the number of trees provides load-balancing: each node forwards only as many messages as it receives. Figure 1 illustrates the distribution trees in a 3-color system.

The membership update in each item must be sent in full along each tree, since it is used to construct the trees. The application data in an item, however, is split into a number of erasure-coded fragments, each of which is forwarded across a different tree. This provides redundancy while imposing minimal bandwidth overhead on the system. With n trees, we use m of n erasure coding, so that all nodes are able to reconstruct the original data even with failures in $n - m$ of the trees. This leads to a bandwidth overhead for application data of close to n/m , with overhead of n for the replicated membership

update. However, as Section 4.4 describes, we are able to eliminate nearly all of this overhead under normal circumstances by suppressing redundant information.

We employ a simple *reconstruction* optimization that provides a substantial improvement in reliability. If a node does not receive the fragment it is supposed to forward, it can regenerate and forward the fragment once it has received m other fragments. This localizes a failure in a given tree to nodes where an ancestor in the current tree failed, and where each parent along the path to the root has experienced a failure in at least $n - m$ trees.

In the case of more than $n - m$ failures, a node may request missing fragments from nodes chosen randomly from its membership view. Section 6.2.1 shows such requests are unnecessary with up to 20% failed nodes.

4.2 Building Trees within a Region

In this section, we describe the algorithm Census uses to build trees. The algorithm must be a deterministic function of the system membership. We use a relatively straightforward algorithm that our experiments show is both computationally efficient and effective at offering low-latency paths, but more sophisticated algorithms are possible at the cost of additional complexity. We first describe how the tree is built in a one-region system; Section 4.3 extends this to multiple regions.

The first step is to color each node, *i.e.* assign it to a tree. This is accomplished by sorting nodes in the region by their ID, then coloring them round-robin, giving an even distribution of colors. Each node then computes all trees, but sends data only on its own tree.

The algorithm uses a hierarchical decomposition of the network coordinate space to exploit node locality. We describe how we build the red tree; other trees are built similarly. The tree is built by recursively subdividing the coordinate space into F sub-regions (where F is the fan-out, typically equal to the number of trees). This is performed by repeatedly splitting sub-regions through the centroid, across their widest axis. One red node from each sub-region is chosen to be a child of the root, and the process continues within each sub-region for the subtree rooted at each child, fewer than F red nodes remain in each sub-region. Figure 2 illustrates the hierarchical decomposition of regions into trees for a fan-out of 4.

Once all red nodes are in the tree, we add the nodes of other colors as leaves. We iterate over the other-colored nodes in ID order, adding them to the red node with free capacity that minimizes the distance to the root via that parent. Our implementation allows nodes to have a fan-out of up to $2F$ when joining leaf nodes to the internal trees, allowing us to better place nodes that are in a sub-region where there is a concentration of a particular color.

As mentioned in Section 2, we use coordinates consisting of two dimensions plus a *height vector* [11]. Height is

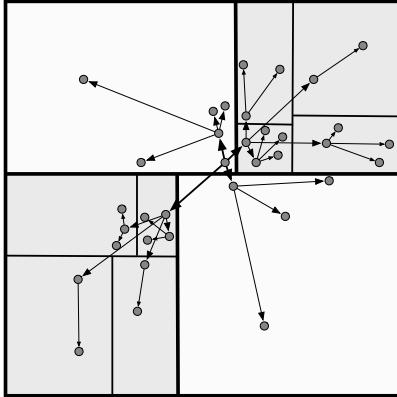


Figure 2: Hierarchical subdivision used to build interior tree with fan-out 4. Each region with more than 4 members is recursively split into smaller sub-regions.

ignored when splitting regions, since it does not reflect the geographic locality of nodes. However, it is used when computing the distance between two nodes, such as when picking the root of a sub-region.

4.3 Building Multi-Region Trees

In the multi-region system, we build an inter-region tree of each color. The nodes in the inter-region tree then serve as roots of the intra-region trees of that color.

The inter-region tree of a particular color is composed of one representative of that color from each region. The representatives are computed by the leader in a global knowledge system, and specified in the summaries in a partial knowledge system. The representatives can be thought of as forming their own “super” region, and we build the tree for that region using recursive subdivision as within a region. The only difference is that we use a smaller fan-out parameter for the inter-region tree, because each node in that tree also acts as a root for a tree within its region, and therefore has descendants in that region as well as descendants in the inter-region tree.

As mentioned in Section 3.3, representatives in the partial knowledge deployment are responsible for propagating the full report that underlies the summary to the members of the region. The extra information is added to the item by the root node of each tree for the region, and thus reaches all the nodes in the region.

4.4 Reducing Bandwidth Consumption

Using erasure coded fragments allows Census to provide high reliability with reasonably low overhead, but is not without bandwidth overhead altogether. In a configuration where 8 out of 16 fragments are required to reconstruct multicast data, each node sends twice as many fragments

as strictly required in the non-failure case. Furthermore, membership updates are transmitted in full on *every* tree, giving even greater overhead.

We minimize bandwidth overhead by observing that redundant fragments and updates are necessary only if there is a failure. Instead of having each parent always send both a membership update and fragment, we designate only one parent per child to send the update and m parents per child to send the fragment. The other parents instead send a short “ping” message to indicate to their child that they have the update and fragment. A child who fails to receive the update or sufficient fragments after a timeout requests data from the parents who sent a ping.

This optimization has the potential to increase latency. Latency increases when there are failures, because a node must request additional fragments from its parents after a timeout. Even without failures, a node must wait to hear from the m parents that are designated to send a fragment, rather than just the first m parents that it hears from.

Fortunately, we are able to exploit membership knowledge to optimize latency. Each parent uses network coordinates to estimate, for each child, the total latency for a message to travel from the root of each tree to that child. Then, it sends a fragment only if it is on one of the m fastest paths to that child. The estimated latencies are also used to set the timeouts for requesting missing fragments. This optimization is possible because Census provides a globally consistent view of network coordinates. Section 6.2.3 shows that it eliminates nearly all redundant bandwidth overhead without greatly increasing latency.

5 Fault Tolerance

In this section we discuss how node and network failures are handled. We consider both crash failures and Byzantine failures, where nodes may behave arbitrarily.

5.1 Crash Failures

Census masks failures of the global leader using replication. A group of $2f_{GL} + 1$ nodes is designated as the *global leader group*, with one member acting as the global leader. Here, f_{GL} is not the maximum number of faulty nodes in the entire system, but rather the number of faulty nodes in the particular leader group; thus the group is relatively small. The members of the leader group use a consensus protocol [25, 21] to agree on the contents of each item: the leader forwards each item to the members of the global leader group, and waits for f_{GL} acknowledgments before distributing it on the multicast trees. The members of the global leader group monitor the leader and select a new one if it appears to have failed.

Tolerating crashes or unexpected node departures is relatively straightforward. Each parent monitors the liveness of its children. If the parent does not receive an acknowledgment after several attempts to forward an item,

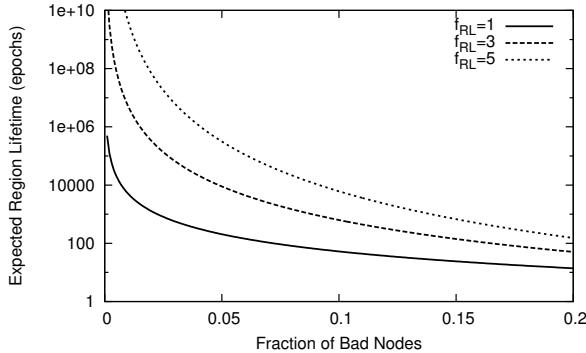


Figure 3: Expected time until a bad leader group (leader and f leader group members faulty) is chosen, in epochs

it reports the absence of the child to the region leader. The child will be removed from the system in the subsequent membership update; if it was only temporarily partitioned from the network, it can rejoin the region. In each epoch, the parent of a particular color is designated as monitor, to prevent multiple parents from reporting the same failure.

Region leaders are not replicated in the global knowledge system; a new region leader is chosen in each epoch, so a failed leader only causes updates to be delayed until it is replaced. In the partial knowledge system, however, we must also ensure that if a region leader sends a summary to the global leader, the corresponding report survives, even if the region leader fails; otherwise, the system would be in an inconsistent state. Census uses a region leader group of $2f_{RL} + 1$ nodes to solve this problem. The region leader sends the report to members of its leader group and waits for f_{RL} acknowledgments before sending the summary to the global leader. Thus, if the representatives receive an item containing a summary for their region, they are guaranteed to be able to retrieve the corresponding report from the leader group, even if the region leader failed, provided that no more than f_{RL} members of the leader group have failed.

Census can select leader groups at random from the system membership, using a deterministic function of the epoch number and region membership. If this approach is used, each summary in the partial knowledge system announces the region’s next leader group, and the global leader group is chosen deterministically from the nodes in the region leader groups.

The size of the leader groups (*i.e.* the values of f_{RL} and f_{GL}) depends on the fraction of nodes expected to be failed *concurrently*, since faulty nodes are removed from the system. Figure 3 shows the expected number of leader groups that can be chosen before choosing a bad group. Because Census detects and removes crashed nodes within a couple of epochs, we can expect the fraction of failed nodes to remain small (*e.g.* under 1%), and

therefore a small value for f will be sufficient even in a very long lived system.

Many applications have some infrastructure nodes that are expected to be very reliable. If so, using these as replicas in leader groups, especially for the global leader, can provide even better reliability. Using infrastructure nodes is particularly well-suited for applications that send multicast data, since they may benefit from having the global leader co-located with the source of multicast data.

5.2 Byzantine Failures

Our solution for Byzantine fault tolerance builds on the approaches used for crash failures, with the obvious extensions. For example, we require signed reports from $f_M + 1$ parents monitoring a failed node to remove it. If this exceeds the number of trees, the node’s predecessors in the region ID space provide additional reports.

We use region leader groups in both the global knowledge and partial knowledge deployments. Since bad nodes may misbehave in ways that cannot be proven, and thus may not be removed from the system, all architectures such as ours must assume the fraction of Byzantine nodes is small. Figure 3 shows that this requirement is fairly constraining if we want the system to be long-lived. For example, with $f = 5$, we must assume no more than 3% faulty nodes to achieve an expected system lifetime of 10 years (with 30-second epochs). Therefore, it would be wise to choose leader groups from infrastructure nodes.

The size of a region leader group is still only $2f_{RL} + 1$, since the group does not run agreement. Instead, a region leader obtains signatures from $f_{RL} + 1$ leader group members, including itself, before sending a summary or report to the global leader. These signatures certify that the group members have seen the updates underlying the report or summary. If the leader is faulty, it may not send the report or summary, but this absence will be rectified in subsequent epochs when a different leader is chosen.

To ensure that a faulty region leader cannot increase the probability that a region leader group contains more than f_{RL} faulty nodes, we choose leader group members based on their IDs, using a common technique from peer-to-peer systems [18, 36]: the first $2f_{RL} + 1$ nodes with IDs greater than the hash of the epoch number (wrapping around if necessary) make up the leader group. A Byzantine region leader cannot invent fictitious joins or departures, because these are signed, and therefore it has no way to control node IDs. It might selectively process join and departure requests in an attempt to control the membership of the next leader group, but this technique is ineffective.

We increase the size of the global leader group to $3f_{GL} + 1$ nodes. The global group runs a Byzantine agreement protocol [9] once per epoch to agree on which summaries will be included in the next item. The next item includes $f_{GL} + 1$ signatures, ensuring that the pro-

tocol ran and the item is valid. The group members also monitor the leader and carry out a view change if it fails. We have developed a protocol that avoids running agreement but requires $2f_{GL} + 1$ signatures, but omit it due to lack of space. Because the failure of the global leader group can stop the entire system, and the tolerated failure level is lower, it is especially important to use trusted infrastructure nodes or other nodes known to be reliable.

5.2.1 Ganging-Up and Duplicates

Two new issues arise because of Census’s multi-region structure. The first is a *ganging-up attack*, where a disproportionate number of Byzantine nodes is concentrated in one region. If so, the fraction of bad nodes in the region may be too high to ensure that region reports are accurate for any reasonable value of f_{RL} . This may occur if an attacker controls many nodes in a particular location, or if Byzantine nodes manipulate their network coordinates to join the region of their choice.

The second problem is that bad nodes might join many regions simultaneously, allowing a small fraction of bad nodes to amplify their population. Such *duplicates* are a problem only in the partial knowledge deployment, where nodes do not know the membership of other regions.

To handle these problems, we exploit the fact that faulty nodes cannot control their node ID. Instead of selecting a region’s leader group from the region’s membership, we select it from a subset of the global membership: we identify a portion of the ID space, and choose leaders from nodes with IDs in this partition. IDs are not under the control of the attacker, so it is safe to assume only a small fraction of nodes in this partition are corrupt, and thus at most f_{RL} failures will occur in a leader group.

Nodes in the leader partition are globally known, even in the partial knowledge system: when a node with an ID in the leader partition joins the system, it is reported to the global leader and announced globally in the next item. These nodes are members of their own region (based on their location), but may also be assigned to the leader group for a different region, and thus need to track that region membership state as well. Nodes in the leader partition are assigned to the leader groups of regions, using consistent hashing, in the same way values are assigned to nodes in distributed hash tables [36]. This keeps assignments relatively stable, minimizing the number of state transfers. When the leader group changes, new members need to fetch matching state from $f_{RL} + 1$ old members.

To detect duplicates in the partial knowledge system, we partition the ID space, and assign each partition to a region, again using consistent hashing. Each region tracks the membership of its assigned partition of the ID space. Every epoch, every region leader reports new joins and departures to the regions responsible for the monitoring the appropriate part of the ID space. These communications

must contain $f_{RL} + 1$ signatures to prevent bad nodes from erroneously flagging others as duplicates. The leader of the monitoring region reports possible duplicates to the regions that contain them; they confirm that the node exists in both regions, then remove and blacklist the node.

5.3 Widespread Failures and Partitions

Since regions are based on proximity, a network partition or power failure may affect a substantial fraction of nodes within a particular region. Short disruptions are already handled by our protocol. When a node recovers and starts receiving items again, it will know from the epoch numbers that it missed some items, and can recover by requesting the items in question from other nodes.

Nodes can survive longer partitions by joining a different region. All nodes know the epoch duration, so they can use their local clock to estimate whether they have gone too many epochs without receiving an item. The global leader can eliminate an entire unresponsive region if it receives no summary or report for many epochs.

6 Evaluation

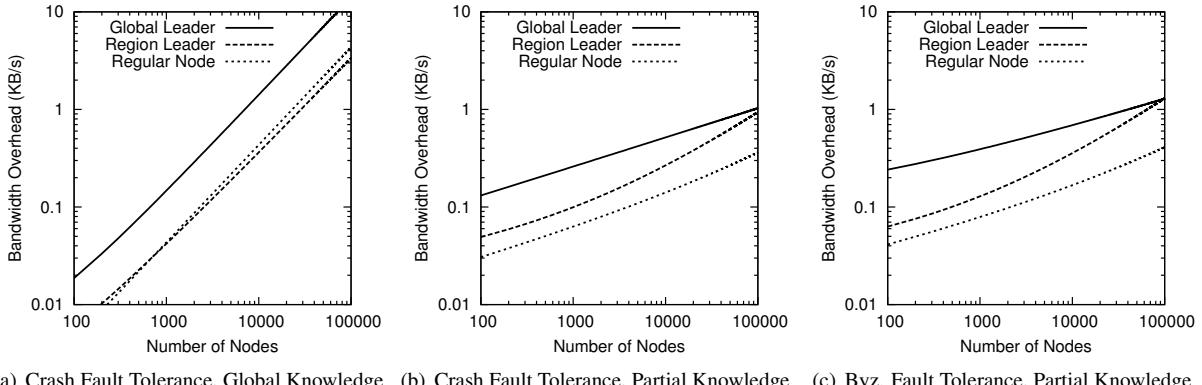
This section evaluates the performance of our system. We implemented a prototype of Census and deployed it on PlanetLab to evaluate its behavior under real-world conditions. Because PlanetLab is much smaller than the large-scale environments our system was designed for, we also examine the reliability, latency, and bandwidth overhead of Census using simulation and theoretical analysis.

6.1 Analytic Results

Figure 4 presents a theoretical analysis of bandwidth overhead per node for a multi-region system supporting both fail-stop and Byzantine failures. The analysis used 8 trees and an epoch interval of 30 seconds. Our figures take all protocol messages into consideration, including UDP overhead, though Figure 4(c) does not include support for preventing ganging-up or for duplicate detection.

Bandwidth utilization in Census is a function of both system membership and churn. These results represent a median node lifetime of 30 minutes, considered a high level of churn with respect to measurement studies of the Gnutella peer-to-peer network [33]. This serves as a “worst-case” figure; in practice, we expect most Census deployments (*e.g.* those in data center environments) would see far lower churn.

The results show that overhead is low for all configurations except when operating with global knowledge on very large system sizes (note the logarithmic axes). Here the global leader needs to process all membership updates, as well as forward these updates to all 8 distribution trees. The other nodes in the system have lower overhead because they forward updates on only one tree. The overhead at the global leader is an order of magni-



(a) Crash Fault Tolerance, Global Knowledge (b) Crash Fault Tolerance, Partial Knowledge (c) Byz. Fault Tolerance, Partial Knowledge

Figure 4: Mean bandwidth overhead with high churn. 30 minute session lengths, 30 second epochs, 8 trees and $f = 3$.

tude lower in the partial knowledge case, where it only receives and distributes the compact region summaries.

In the partial knowledge cases (Figures 4(b) and 4(c)) the region leader incurs more overhead than the regular nodes, primarily due to forwarding each report to the leader group and representatives before sending a summary to the global leader. Supporting Byzantine fault tolerance imposes little additional overhead for the region leaders and global leader, because the cost of the additional signatures and agreement messages are dominated by the costs of forwarding summaries and reports.

These results are sensitive to region size, particularly in large deployments, as this affects the trade-off between load on the region leaders and on the global leader. For the purpose of our analysis we set the number of regions to $\sqrt[3]{n}$ nodes, mimicking the proportions of a large-scale deployment of 100 regions each containing 10,000 nodes.

6.2 Simulation Results

We used a discrete-event simulator written for this project to evaluate reliability, latency, and the effectiveness of our selective fragment transmission optimization. The simulator models propagation delay between hosts, but does not model queuing delay or network loss; loss due to bad links is represented by overlay node failures.

Two topologies were used in our simulations: the King topology, and a random synthetic network topology. The King topology is derived from the latency matrix of 1740 Internet DNS servers used in the evaluation of Vivaldi [11], collected using the King method [15]. This topology represents a typical distribution of nodes in the Internet, including geographical clustering. We also generate a number of synthetic topologies, with nodes uniformly distributed within the coordinate space. These random topologies allow us to examine the performance of the algorithm when nodes are not tightly clustered into regions, and to freely experiment with network sizes

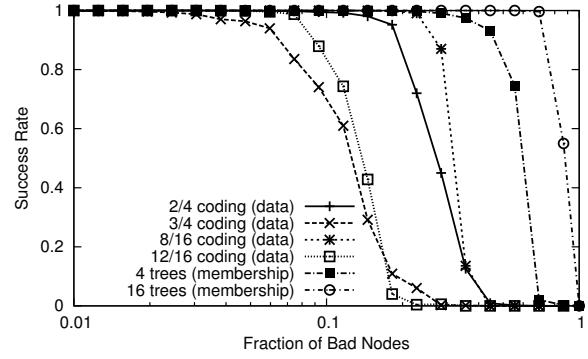


Figure 5: Fraction of nodes that receive tree data in the presence of faulty nodes, with 10 regions of 1,000 nodes.

without affecting the distribution of nodes.

While our simulator measures network delays using latencies, our algorithms operate solely in the coordinate domain. We generated coordinates from the King data using a centralized version of the Vivaldi algorithm [11]. Coordinates consist of two dimensions and a height vector, as was found to effectively model latencies in Vivaldi. These coordinates do not perfectly model actual network delays, as discussed in Section 6.2.2.

6.2.1 Fault Tolerance

Figure 5 examines the reliability of our distribution trees for disseminating membership updates and application data under simulation. In each experiment we operate with 10 regions of 1,000 nodes each; single-region deployments see equivalent results. The reliability is a function only of the tree fan-out and our disjointness constraint, and does not depend on network topology.

The leftmost four lines show the fraction of nodes that are able to reconstruct application data under various erasure coding configurations. Census achieves very high

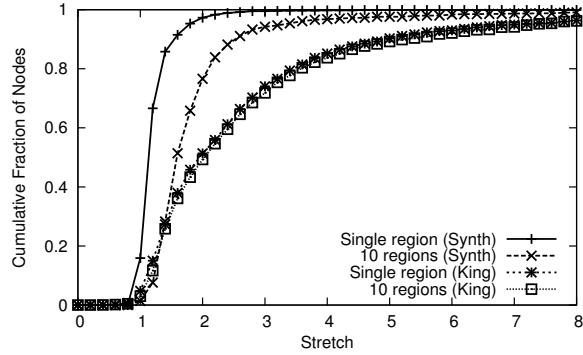


Figure 6: CDF of multicast transmission stretch in King and 10,000 node synthetic topologies. 4/8 erasure coding.

success rates with up to 30% failed nodes in the 16-tree configuration, with 2 \times redundancy (8/16 coding). This high failure rate is unlikely, since our membership management protocol removes failed nodes promptly. The primary factor influencing reliability is the redundancy rate. Increasing the number of trees also improves reliability, even with the same level of redundancy: using 16 trees instead of 4 tolerates nearly 10% more failures. Additional trees improve reliability by reducing the probability of $n - m$ parents failing, but comes with the cost of more messages and more representatives to maintain.

The rightmost two lines show the fraction of nodes that receive membership update information, which is sent in full on each tree. In a 16-tree deployment, we find that every non-faulty node receives membership information on at least one tree, even if as many as 70% of the region members have failed. Even on a 4-tree deployment, all nodes receive membership information in the presence of upwards of 20% failed nodes.

The high reliability exhibited in Figure 5 is partially due to our reconstruction optimization, discussed in Section 4.1. An 8/16 deployment using reconstruction allows all non-faulty nodes to receive application data with as many as 22% failed nodes, but tolerates only 7.5% failures without reconstruction. Reconstruction mitigates the effects of a failure by allowing a tree to heal below a faulty node, using fragments from other trees.

6.2.2 Latency

Census’s multicast mechanism must not impose excessive communication delay. We evaluate this delay in terms of *stretch*, defined as the total time taken for a node to receive enough fragments to reconstruct the data, divided by the unicast latency between the server and the node.

Figure 6 shows stretch on both the King and synthetic topologies, assuming no failures and using 8 trees; results for 16 trees are similar. The figure shows that stretch

is close to 1 on the synthetic topology, indicating that our tree-building mechanism produces highly efficient trees. Stretch is still low on the King topology, at an average of 2, but higher than in the synthetic topology. This reflects the fact that the coordinates generated for the King topology are not perfect predictors of network latency, while the network coordinates in the synthetic topology are assumed perfect. The small fraction of nodes with stretch below 1 are instances where node latencies violate the triangle inequality, and the multicast overlay achieves lower latency than unicast transmission.

Stretch is slightly higher in the multi-region deployment with the synthetic topology because the inter-region tree must be constructed only of representatives. In the synthetic topology, which has no geographic locality, stretch increases because the representatives may not be optimally placed within each region. However, this effect is negligible using the King topology, because nodes within a region are clustered together and therefore the choice of representatives has little effect on latency.

Our stretch compares favorably with existing multicast systems, such as SplitStream [7], which also has a stretch of approximately 2. In all cases, transmission delay overhead is very low compared to typical epoch times.

6.2.3 Selective Fragment Transmission

Figure 7(a) illustrates the bandwidth savings of our optimization to avoid sending redundant fragments (described in Section 4.4), using 2 \times redundancy and 8 trees on the King topology. In the figure, bandwidth is measured relative to the baseline approach of sending all fragments. We see a 50% reduction in bandwidth with this optimization; overhead increases slightly at higher failure rates, as children request additional data after timeouts.

Figure 7(b) shows this optimization’s impact on latency. It adds negligible additional stretch at low failure rates, because Census chooses which fragments to distribute based on accurate predictions of tree latencies. At higher failure rates, latency increases as clients are forced to request additional fragments from other parents, introducing delays throughout the distribution trees.

The figures indicate that the optimization is very effective in the expected deployments where the failure rate is low. If the expected failure rate is higher, sending one extra fragment reduces latency with little impact on bandwidth utilization.

6.3 PlanetLab Experiments

We verify our results using an implementation of our system, deployed on 614 nodes on PlanetLab. While this does not approach the large system sizes for which our protocol was designed, the experiment provides a proof of concept for a real widespread deployment, and allows

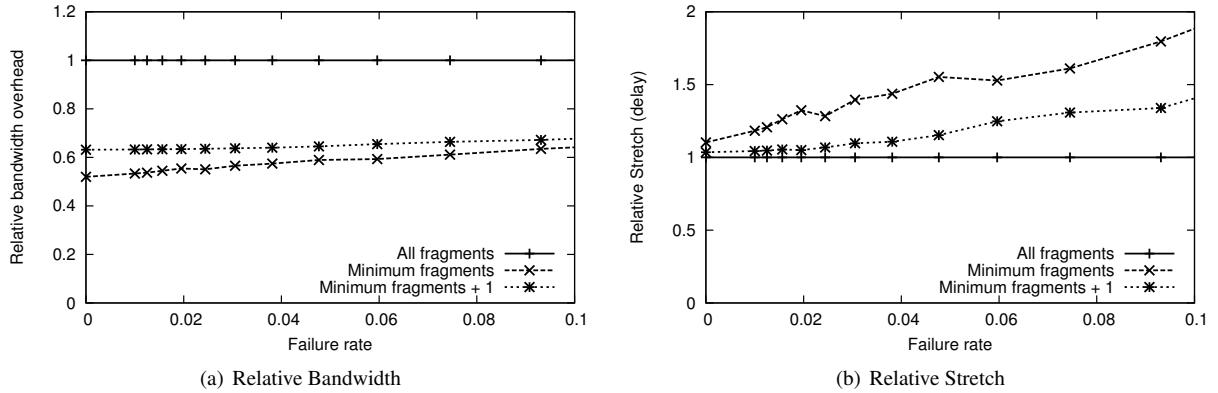


Figure 7: Performance impact of avoiding redundant fragments in multicast trees. (4 of 8 coding)

us to observe the system under realistic conditions such as non-uniform node failures.

Our implementation supports multiple regions, with dynamic splits and joins, but we found that a single region was sufficient for the number of nodes in our PlanetLab deployment, and more representative of region sizes that would be seen in larger deployments. The implementation currently supports fail-stop failures, moving the leader each epoch, but does not tolerate Byzantine failures.

We configured the system to use 6 distribution trees, with an epoch time of 30 seconds. In addition to membership information, we distributed a 1 KB application message each epoch using 3-of-6 erasure coding, to test the reliability and overhead of this part of our system.

We ran Census for 140 epochs of 30 seconds each. As indicated in Figure 8(a), during the experiment, we failed 10% of the nodes simultaneously, then restarted them; we then did the same with 25% of the nodes. The graph shows the number of nodes reported in our system's membership view. Census reacts quickly to the sudden membership changes; the slight delay reflects the time needed for parents to decide that their children are faulty.

Figure 8(b) shows the average total bandwidth usage (both upstream and downstream) experienced by nodes in our system. Each node uses about 0.1 KB/s at steady-state, much of which is due to the size of the multicast data; the shaded region of the graph represents the theoretical minimum cost of disseminating a 1 KB message each epoch. Bandwidth usage increases for a brief time after our sudden membership changes, peaking at 0.9 KB/s immediately after 25% of the nodes rejoin at once. Node rejoins are more costly than node failures, because more information needs to be announced globally for a newly-joined node and the new node needs to obtain the system membership. We have also run the system for much longer periods, with similar steady-state bandwidth usage.

7 Applications

Knowledge of system membership is a powerful tool that can simplify the design of many distributed systems. An obvious application of Census is to support administration of large multi-site data centers, where Byzantine failures are rare (but do occur), and locality is captured by our region abstraction. Census is also useful as an infrastructure for developing applications in such large distributed systems. In this section, we describe a few representative systems whose design can be simplified with Census.

7.1 One-Hop Distributed Hash Tables

A distributed hash table is a storage system that uses a distributed algorithm, usually based on consistent hashing [18], to map item keys to the nodes responsible for their storage. This abstraction has proven useful for organizing systems at scales ranging from thousands of nodes in data centers [12] to millions of nodes in peer-to-peer networks [23]. The complexity in such systems lies primarily in maintaining membership information to route requests to the correct node, a straightforward task with the full membership information that Census provides.

Most DHTs do not maintain full membership knowledge at each host, so multiple (*e.g.* $O(\log N)$) routing steps are required to locate the node responsible for an object. Full global knowledge allows a message to be routed in one step. In larger systems that require partial knowledge, messages can be routed in two steps. The key now identifies both the responsible region and a node within that region. A node first routes a message to any member of the correct region, which then forwards it to the responsible node, much like the two-hop routing scheme of Gupta et al. [16]. Although our membership management overhead does not scale asymptotically as well as many DHT designs, our analysis in Section 6.1 shows that the costs are reasonable in most deployments.

From a fault-tolerance perspective, Census's member-

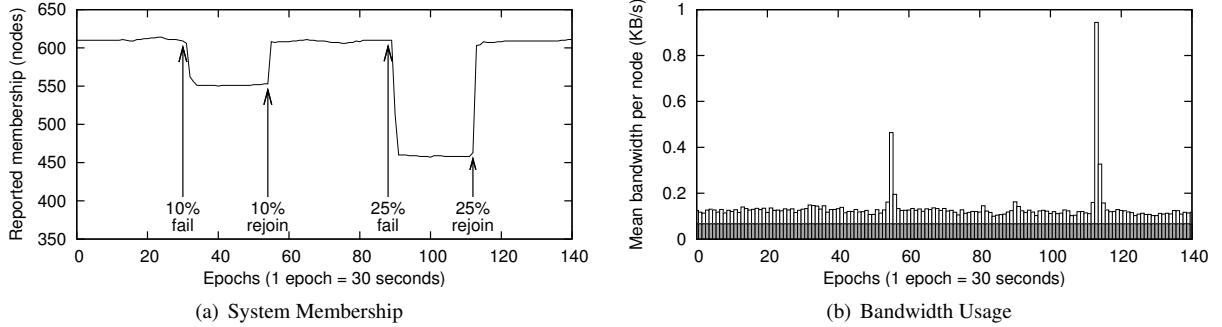


Figure 8: Results of a 614-node, 70-minute PlanetLab deployment with 10% and 25% correlated failures.

ship views provide several advantages. Single-hop routing eliminates the possibility of a malicious intermediate node redirecting a request [35]. The fault-tolerance of our protocol prevents Eclipse attacks, where malicious nodes influence an honest node’s routing table [6]. Applications that use replication techniques to ensure consistency across replicas benefit from Census’s consistent membership views, since all nodes in the system agree on the identity of the replicas for each epoch.

Finally, our use of full membership information can enable more sophisticated placement algorithms than the standard consistent hashing approach. Any deterministic function of the membership view, including location information, suffices. For example, we might choose one replica for a data item based on its key, and the others based on location: either nearby nodes (for improved performance) or distant ones (for failure-independence).

7.2 Application-Layer Multicast

Census allows applications to disseminate information on multicast trees by piggybacking it on items. The application can do this as needed: occasionally, on every item, or more frequently. An example of where more frequent multicast is needed is to broadcast video. For high-bandwidth multicast like video streaming, the costs of maintaining membership become less significant.

Compared to other scalable multicast systems, Census’s multicast trees can provide higher reliability, using optimizations like reconstruction and selective fragment transmission, and can tolerate Byzantine behavior. The availability of consistent membership views keeps the multicast protocol relatively simple, while still providing strong performance and reliability guarantees.

Census can also be used to construct a publish-subscribe system where only certain nodes are interested in receiving each message. One node is designated as responsible for each interest group, and other nodes contact it to publish or subscribe. When this node receives a message to distribute, it constructs multicast trees over

just the subscribers, using them to disseminate both the message and changes in subscriber membership. This multicast is independent of the one we use to distribute membership information, but the trees can be constructed using the same algorithm.

7.3 Cooperative Caching

We are currently developing a wide-scale storage application where a small set of nodes act as servers, storing the definitive copy of system data. The other nodes in the system are clients. To perform operations, they fetch pages of data from the server into local caches and execute operations locally; they write back modified pages to the server when the computation is finished.

To reduce load on the storage servers, clients share their caches, fetching missing pages from nearby clients. A partial knowledge Census deployment makes it easy for clients to identify other nearby clients. We are investigating two approaches to finding pages. In one, nodes announce pages they are caching on the multicast tree within a region, so each node in the region always knows which pages are cached nearby. The other uses an approach similar to peer-to-peer indexing systems (*e.g.* [10]): we use consistent hashing [18] to designate for each page one node per region that keeps track of which region members have that page cached. Members register with this node once they have fetched a page, and check with it when they are looking for a page.

Cached information inevitably becomes stale, rendering it useless for computations that require consistency. To keep caches up to date, storage servers in this system use Census’s multicast service to distribute an *invalidation stream*. This consists of periodic notices listing the set of recently modified pages; when a node receives such a notice, it discards the invalid pages from its cache.

8 Related Work

There is a long history of research in group communication systems, which provide a multicast abstraction along

with a membership management service [14, 37, 19, 2, 26, 29]. Many of these systems provide support for group communication while maintaining *virtual synchrony* [3], a model similar to our use of epochs to establish consistent views of system information. Such systems are typically not designed to scale to large system populations, and often require dedicated membership servers, which do not fit well with our decentralized model.

Spread [2] and ISIS [4] use an abstraction of many lightweight membership groups mapping onto a smaller set of core groups, allowing the system to scale to large numbers of multicast groups, but not large membership sizes. We take a different approach in using regions to group physical nodes, and scale to large system memberships, without providing a multiple-group abstraction. Quicksilver [26] aims to scale in both the number of groups and the number of nodes, but does not exploit our physical hierarchy to minimize latency and communication overhead in large system deployments.

Prior group communication systems have also aimed to tolerate Byzantine faults, in protocols such as Rampart [30] and SecureRing [20]. Updating the membership view in these systems requires executing a three-phase commit protocol across all nodes, which is impractical with more than a few nodes. By restricting our protocol to require Byzantine agreement across a small subset of nodes, we achieve greater scalability. Rodrigues proposed a membership service using similar techniques [31], but it does not provide locality-based regions or partial knowledge, and assumes an existing multicast mechanism.

Many large-scale distributed systems employ ad-hoc solutions to track dynamic membership. A common approach is to use a centralized server to maintain the list of active nodes, as in Google’s Chubby lock service [5]. Such an approach requires all clients to communicate directly with a replicated server, which may be undesirable from a scalability perspective. An alternative, decentralized approach seen in Amazon’s Dynamo system [12] is to track system membership using a gossip protocol. This approach provides only eventual consistency, which is inadequate for many applications, and can be slow to converge. These systems also typically do not tolerate Byzantine faults, as evidenced by a highly-publicized outage of Amazon’s S3 service [1].

Distributed lookup services, such as Chord [36] and Pastry [32], provide a scalable approach to distributed systems management, but none of these systems provides a consistent view of membership. They are also vulnerable to attacks in which Byzantine nodes cause requests to be misdirected; solving this problem involves trading-off performance for probabilistic guarantees of correctness [6].

Fireflies [17] provides each node with a view of system membership, using gossip techniques that tolerate Byzantine failures. However, it does not guarantee a *consistent*

global membership view, instead giving a probabilistic agreement. Also, our location-aware distribution trees offer faster message delivery and reaction to changes.

Our system’s multicast protocol for disseminating membership updates builds on the multitude of recent application-level multicast systems. Most (but not all) of these systems organize the overlay as a tree to minimize latency; the tree can be constructed either by a centralized authority [28, 27] or by a distributed algorithm [8, 7, 22]. We use a different approach: relying on the availability of global, consistent membership views, we run what is essentially a centralized tree-building algorithm independently at each node, producing identical, optimized trees without a central authority.

SplitStream [7] distributes erasure-coded fragments across multiple interior-node-disjoint multicast trees in order to improve resilience and better distribute load among the nodes. Our overlay has the same topology, but it is constructed in a different manner. We also employ new optimizations, such as selective fragment distribution and fragment reconstruction, which provide higher levels of reliability with lower bandwidth overhead.

9 Conclusions

Scalable Internet services are often built as distributed systems that reconfigure themselves automatically as new nodes become available and old nodes fail. Such systems must track their membership. Although many membership services exist, all current systems are either impractical at large scale, or provide weak semantics that complicate application design.

Census is a membership management platform for building distributed applications that provides both strong semantics and scalability. It provides consistent membership views, following the virtual synchrony model, simplifying the design of applications that use it. The protocol scales to large system sizes by automatically partitioning nodes into proximity-based regions, which constrains the volume of membership information a node needs to track. Using lightweight quorum protocols and agreement across small groups of nodes, Census can maintain scalability while tolerating crash failures and a small fraction of Byzantine-faulty nodes.

Census distributes membership updates and application data using a unconventional multicast protocol that takes advantage of the availability of membership data. The key idea is that the distribution tree structure is determined entirely by the system membership state, allowing nodes to independently compute identical trees. This approach allows the tree to be reconstructed without any overhead other than that required for tracking membership. As our experiments show, using network coordinates produces trees that distribute data with low latency, and the multiple-tree overlay structure provides reliable data dissemination

even in the presence of large correlated failures.

We deployed Census on PlanetLab and hope to make the deployment available as a public service. We are currently using it as the platform for a large-scale storage system we are designing, and expect that it will be similarly useful for other reconfigurable distributed systems.

Acknowledgments

We thank our shepherd Marvin Theimer, Austin Clements, and the anonymous reviewers for their valuable feedback. This research was supported by NSF ITR grant CNS-0428107.

References

- [1] Amazon Web Services. Amazon S3 availability event: July 20, 2008. <http://status.aws.amazon.com/s3-20080720.html>, July 2008.
- [2] Y. Amir and J. Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, The Johns Hopkins University, Baltimore, MD, 1998.
- [3] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. SOSP '87*, Nov. 1987.
- [4] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. In *Transactions on Computer Systems*, volume 9, Aug. 1991.
- [5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI '06*, Seattle, WA, Nov. 2006.
- [6] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. OSDI '02*, Boston, MA, Nov. 2002.
- [7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proc. SOSP '03*, Bolton Landing, NY, 2003.
- [8] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, Oct. 2002.
- [9] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.
- [10] A. T. Clements, D. R. K. Ports, and D. R. Karger. Arpeggio: Metadata searching and content sharing with Chord. In *Proc. IPTPS '05*, Ithaca, NY, Feb. 2005.
- [11] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proc. ACM SIGCOMM 2004*, Portland, OR, Aug. 2004.
- [12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. In *Proc. SOSP '07*, Stevenson, WA, Oct. 2007.
- [13] J. R. Douceur. The Sybil attack. In *Proc. IPTPS '02*, Cambridge, MA, Feb. 2002.
- [14] B. B. Glade, K. P. Birman, R. C. B. Cooper, and R. van Renesse. Lightweight process groups in the ISIS system. *Distributed Systems Engineering*, 1:29–36, 1993.
- [15] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proc. IMW '02*, Marseille, France, Nov. 2002.
- [16] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *Proc. NSDI '04*, San Francisco, CA, Mar. 2004.
- [17] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proc. EuroSys '06*, Leuven, Belgium, Apr. 2006.
- [18] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. STOC '97*, El Paso, TX, May 1998.
- [19] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, 20:191–238, 2002.
- [20] K. Kuhlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proc. HICSS '98*, Hawaii, Jan. 1998.
- [21] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
- [22] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. MON: On-demand overlays for distributed system management. In *Proc. WORLDS '05*, San Francisco, CA, Dec. 2005.
- [23] P. Maymounkov and D. Mazières. Kademia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS '02*, Cambridge, MA, Feb. 2002.
- [24] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *Proc. P2PEcon '04*, Cambridge, MA, June 2004.
- [25] B. Oki and B. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. PODC '88*, 1988.
- [26] K. Ostrowski, K. Birman, and D. Dolev. QuickSilver scalable multicast. In *Proc. NCA '08*, Cambridge, MA, July 2008.
- [27] V. N. Padmanabhan, H. J. Wang, and P. A. Chou. Resilient peer-to-peer streaming. In *Proc. ICNP '03*, Atlanta, GA, Nov. 2003.
- [28] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proc. USITS '01*, San Francisco, CA, Mar. 2001.
- [29] D. Powell, D. Seaton, G. Bonn, P. Veríssimo, and F. Waeselynck. The Delta-4 approach to dependability in open distributed computing systems. In *Proc. FTCS '88*, June 1988.
- [30] M. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, Jan. 1996.
- [31] R. Rodrigues. *Robust Services in Dynamic Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2005.
- [32] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware '01*, Heidelberg, Nov. 2001.
- [33] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN '02*, San Jose, CA, June 2002.
- [34] M. Sherr, B. T. Loo, and M. Blaze. Veracity: A fully decentralized service for securing network coordinate systems. In *Proc. IPTPS '08*, Feb. 2008.
- [35] E. Sit and R. Morris. Security concerns for peer-to-peer distributed hash tables. In *Proc. IPTPS '02*, Cambridge, MA, Feb. 2002.
- [36] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Networking*, 11(1):149–160, Feb. 2003.
- [37] R. van Renesse, K. P. Birman, and S. M. Eis. Horus: A flexible group communication system. *Communications of the ACM*, 39:76–83, 1996.
- [38] D. Zage and C. Nita-Rotaru. On the accuracy of decentralized network coordinate systems in adversarial networks. In *Proc. CCS '07*, Alexandria, VA, Oct. 2007.
- [39] Y. Zhou. Computing network coordinates in the presence of Byzantine faults. Master’s thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2008. Available as technical report MIT-CSAIL-TR-2009-015.
- [40] Y. Zhu, B. Li, and J. Guo. Multicast with network coding in application-layer overlay networks. *IEEE Journal on Selected Areas in Communications*, 22(1), Jan. 2004.

Veracity: Practical Secure Network Coordinates via Vote-based Agreements

Micah Sherr Matt Blaze Boon Thau Loo
University of Pennsylvania

Abstract

Decentralized network coordinate systems promise efficient network distance estimates across Internet end-hosts. These systems support a wide range of network services, including proximity-based routing, neighbor selection in overlays, network-aware overlays, and replica placement in content-distribution networks.

This paper describes *Veracity*, a practical fully-decentralized service for securing network coordinate systems. In Veracity, all advertised coordinates and subsequent coordinate updates must be independently verified by a small set of nodes via a voting scheme. Unlike existing approaches, Veracity does not require any *a priori* secrets or trusted parties, and does not depend on outlier analysis of coordinates based on a fixed set of neighbors. We have implemented Veracity by modifying an open-source network coordinate system, and have demonstrated within a simulated network environment and deployment on PlanetLab that Veracity mitigates attacks for moderate sizes of malicious nodes (up to 30% of the network), even when coalitions of attackers coordinate their attacks. We further show that Veracity resists high levels of churn and incurs only a modest communication overhead.

1 Introduction

Decentralized network coordinate systems such as Vivaldi [7], PIC [6], ICS [19], Big-bang simulation [29], and NPS [22] have been proposed as a means of efficiently estimating network distances without having to explicitly contact the end-hosts involved. Distributed algorithms map nodes to n -dimensional coordinates such that the distance between two nodes' coordinates corresponds to a network distance (e.g., latency) between the pair. For a network with N nodes, coordinate systems linearize the information necessary to compute pairwise

network distances, allowing nodes to estimate N^2 distances using N embedded coordinates.

Coordinate systems support a wide range of network services, including proximity-based routing [31], neighbor selection in overlays [9], network-aware overlays [23], and replica placement in content-distribution networks [8, 37]. Several large-scale coordinate systems are currently deployed on the Internet. For example, the Vuze client (formally called Azureus) for Bit-Torrent uses coordinate systems for efficient distributed hash table (DHT) traversal [2] and locality-based neighbor selection [35], and currently operates on more than one million nodes [18].

Unfortunately, the distributed nature of coordinate systems make them particularly vulnerable to insider manipulation. To illustrate, recent studies [16] on Vivaldi have shown that when 30% of nodes lie about their coordinates, Vivaldi's accuracy decreases by a factor of five. When attackers collude, even 5% malicious nodes have a sizable impact on the system's accuracy.

In addition to causing significantly decreased accuracy and performance, corrupted coordinate systems may serve as stepping stones for attacks against the applications that rely on them. Attackers who control the coordinate system may advertise attractive (but false) coordinates for nodes under their control, increasing the likelihood that such hosts will be selected for routes, neighbors, or replicas. Such compromises enable myriad attacks against the overlying services. For example, malicious nodes may misdirect intercepted messages sent via overlay routing, return false data when serving as a replica in a content distribution network, or partition the keyspace in a distributed hash table.

This paper presents *Veracity*, a *fully* decentralized service for securing network coordinate systems. Veracity provides a practical deployment path while provid-

ing equivalent (or greater) security than previously proposed coordinate security systems. Unlike prior proposals, Veracity does not require either pre-selected trusted nodes [15], the triangle inequality test [6], or outlier detection based on a fixed neighbor set [40], allowing Veracity to be practically deployed and react more rapidly to changes in network conditions. Veracity is also agnostic to the type of decentralized coordinate system being deployed, and can be employed as a protection service over existing decentralized coordinate systems [7, 6, 19, 22].

At a high-level, Veracity utilizes a two-step verification process. The first step involves a majority vote-based scheme in which a published coordinate has to be independently verified by a deterministically assigned set of *verification nodes* before it is used by peers. An adversary who attempts to disrupt the network by publishing inconsistent coordinates will fail this verification step, and consequently its coordinates will be ignored. As an additional measure, a second verification step utilizes a set of randomly chosen peers to independently compute the estimation error due to a new coordinate, and reject the coordinate if the error is above a threshold. This second protection mechanism detects attacks in which malicious nodes delay responses to measurement probes. The combination of the two techniques ensures that Veracity can tolerate a high fraction of malicious nodes that concurrently report false coordinates and delay latency measurements.

In this paper, we focus our implementation and evaluation on Vivaldi since it is widely used [3] and has been the focus of recent work [15, 40] on securing coordinate systems. We demonstrate via execution in a simulated network environment using realistic network traces [38, 17] and a deployment on PlanetLab that Veracity mitigates attacks for moderate sizes of malicious nodes (up to 30% of the network), even when coalitions of attackers coordinate their attacks. We further show that Veracity is resistant to high levels of churn and incurs only a modest communication overhead.

2 Background

In this section, we present a brief introduction to the Vivaldi system and outline threat models and metrics.

2.1 Vivaldi Coordinate System

Vivaldi uses a fully distributed spring relaxation algorithm, requiring no fixed network infrastructure and no distinguished nodes. The system envisions a spring between each pair of nodes, with the resting position of the spring equaling the network latency between the pair. At any point in time, the distance between the nodes in

the coordinate space determines the current length of the spring connecting the nodes.

Nodes adjust their coordinates after collecting published coordinate and latency measurements from a randomly chosen neighbor. Consider a node i that wishes to update its coordinate C_i . It picks a randomly chosen neighbor j , retrieves its coordinate C_j and performs a round-trip measurement RTT_{ij} from itself to j . The squared error function, $E_{ij} = (RTT_{ij} - ||C_i - C_j||)^2$ (where $||C_i - C_j||$ is the distance between their coordinates) denotes the *estimation error* between the coordinates of i and j . Using Vivaldi's spring relaxation algorithm, E_{ij} reflects the potential energy of the spring connecting the two nodes. Vivaldi attempts to minimize the potential energies over all springs. In each timestep of the algorithm, nodes allow themselves to be pulled or pushed by a connected spring. The system converges when the squared error function (i.e., the potential energies) is minimized below a threshold.

2.2 Attacker Model

Prior studies [16, 15] have demonstrated that coordinate systems are susceptible to three classes of attacks: *disorder* attacks in which malicious insiders attempt to decrease the accuracy of the system by advertising false coordinates and delaying RTT responses, and *isolation* and *repulsion* attacks in which the attacker respectively attempts to isolate or repulse a subset of targeted nodes. Veracity's general approach defends against malicious nodes that falsify their coordinates or induce/report artificially inflated latencies. Hence, the techniques described in this paper can mitigate all three attacks.

We adopt the constrained-collusion Byzantine model proposed by Castro *et al.* [5] in which malicious nodes can insert, delete, or delay messages. Given a network of size N , and some fraction ($f < 1$) of malicious attackers, there exist independent coalitions of size cN , where $1/N \leq c \leq f$.

2.3 Metrics

To assess the accuracy of a virtual coordinate, we measure the **median error ratio** of a node n_i , defined as the median over the *error ratios*

$$\frac{|RTT(n_i, n_j) - ||C_{n_i} - C_{n_j}|||}{RTT(n_i, n_j)} \quad (1)$$

between n_i and all other nodes n_j ($n_i \neq n_j$). Conceptually, Equation 1 computes the difference between the computed latency between n_i and n_j based on coordinates ($C_{n_i} - C_{n_j}$) and the actual measured RTT (denoted by $RTT(n_i, n_j)$). The accuracy of a node's coordinate

increases inversely with its median error ratio. The use of median provides an intuitive measure of a coordinate’s accuracy and is more robust than average to the effects of outlier errors. Previous approaches [7, 22, 40] define similar metrics.

To gauge the accuracy of the system as a whole, we define the **system error ratio** as the median over all peers’ median error ratios. The system error ratio enables us to quantitatively compare the performance and security of Veracity and Vivaldi. To show lower performance bounds, we also consider the **90th percentile error ratio** – i.e., the 90th percentile of nodes’ median error ratios.

3 Overview of Veracity

We first provide an overview of Veracity’s security mechanisms. We base our description on Vivaldi’s coordinate update model. While other decentralized coordinate systems [6, 19, 22] differ in their implementations, the update models are conceptually similar to Vivaldi’s, and hence, Veracity’s techniques are applicable to these systems as well.

To update its coordinate, a participating node (the *investigator*) periodically obtains the coordinate of a selected peer (the *publisher*) and measures the RTT between the two nodes. In most implementations, the publisher is typically a pre-assigned neighbor node of the investigator. In Veracity, we relax the requirement that a publisher has to be a fixed neighbor of the investigator and instead use a distributed directory service (Section 4.2) to enable investigators to scalably select random publishers on demand.

The basic update model of Vivaldi leads to two possible avenues of attacks: first, if the publisher is dishonest, it may report inaccurate coordinates. Second, the publisher may delay the RTT probe response to increase the error of the investigator’s updated coordinate. To defend against such attacks, Veracity protects the underlying coordinate system through a two-step verification process in which groups of nodes independently verify the correctness of another node’s coordinates. We outline these two processes below, with additional details presented in Section 4.

- **Publisher coordinate verification:** When an investigator requests a coordinate from a publisher, a deterministic set of peers called the *verification set* (*VSet*) verifies the publisher’s claimed coordinate. Veracity assigns each publisher a unique VSet. Each VSet member independently assesses the accuracy of the coordinate by conducting its own empirical measurements to the publisher and computes the coordinate’s estimation

error. If a majority of the VSet does not accept the publisher’s coordinate, the investigator discards the coordinate.

- **Candidate coordinate verification:** Once an investigator verifies the publisher’s coordinate, it proceeds to update its own coordinate based on its empirical RTT measurement between itself and the publisher. To detect cases in which the publisher purposefully delays the RTT probe response, the investigator updates its coordinate to a new one only if the new coordinate results in no more than a small increase in estimation error computed amongst an independent and *randomly chosen* set of peers (the *RSet*).

An important benefit of Veracity is that it makes no distinction between intentionally falsified coordinates and those that are inaccurate due to limitations of the coordinate embedding process. In either case, Veracity prevents the use of inaccurate coordinates.

4 Veracity Verification Protocols

This section presents details of Veracity’s two-step verification protocol. We first focus on various mechanisms necessary to realize *publisher coordinate verification* that prevents investigators from considering inaccurate coordinates. We then motivate and describe the *candidate coordinate verification* that protects against malicious RTT probe delays by the publisher.

4.1 VSet Construction

When a Veracity node joins the network, it computes a *globally unique identifier* (GUID) by applying a collision-resistant cryptographic hash function H (e.g., SHA-1) to its network address. (To prevent malicious peers from strategically positioning their GUIDs, Veracity restricts allowable port numbers to a small range.) Given a node with GUID g , the members of its VSet are the peers whose GUIDs are closest to h_1, \dots, h_Γ , determined using the recurrence:

$$h_i = \begin{cases} H(g) & \text{if } i = 1 \\ H(h_{i-1}) & \text{if } i > 1 \end{cases} \quad (2)$$

where i ranges from 1 to the *VSet size*, Γ . A larger Γ increases the trustworthiness of coordinates (since more nodes are required in the verification process) at the expense of additional communication.

VSet construction utilizes a hash function to increase the difficulty of stacking VSets with collaborating malicious nodes. Attackers who control large coalitions of peers may be able to populate a majority of a particular malicious node’s VSet (for example, by strategically choosing IP addresses within its assigned range),

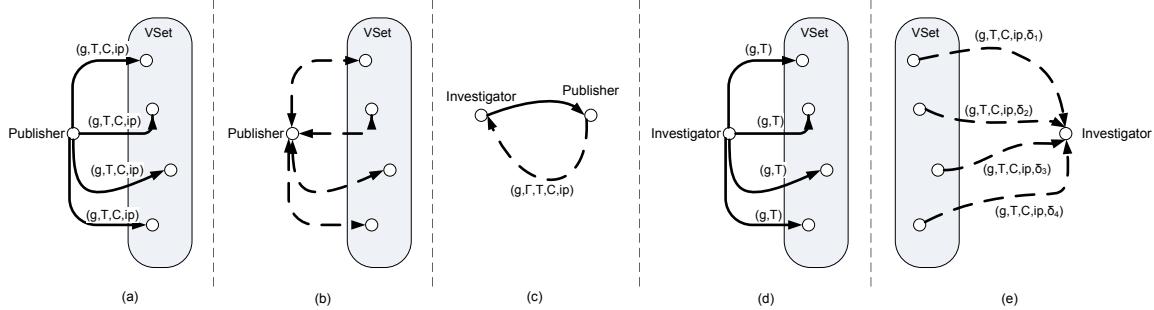


Figure 1: Publisher coordinate verification. Solid lines denote messages sent via `deliver` and dotted lines represent messages sent via direct IP. (a) Publisher distributes update tuple to VSet members using `deliver` messages addressed to GUIDs based on recursive hashes. (b) VSet members measure the RTT between themselves and Publisher. (c) Investigator queries Publisher and Publisher responds with claim tuple. (d) Investigator sends evidence query to Publisher’s VSet members. (e) VSet members send evidence tuples to investigator.

but such VSet stacking requires at a minimum $\lceil \frac{\Gamma}{2} \rceil$ peers per VSet. In practice, many more malicious peers are required since the attacker does not have complete discretion over the IP addresses of its coalition members. Moreover, as nodes join and leave the network, VSet members change (since the nodes whose GUIDs are closest to h_1, \dots, h_Γ also change), significantly impairing the ability to persistently stack VSets.

4.2 Locating and Updating VSet Members

Veracity utilizes a *distributed directory service* to resolve VSet members and route messages based on node GUIDs. The directory service implements a single API function, `deliver` (g, m), which delivers the message m to the peer whose GUID is closest to g according to a keyspace distance metric. Veracity is compatible with any distributed directory service that supports the `deliver` function. We explore the implementation of distributed directory services in Section 5.

As shown in Figure 1(a), when a publisher updates its coordinate, it transmits an *update tuple* (g, τ, C, ip) to members of its VSet using the `deliver` function provided by the directory service. The update tuple contains the following values: g is the publisher’s GUID, τ is a logical timestamp incremented whenever the publisher updates his coordinate, C is the new coordinate, and ip is the publisher’s network address. Upon receiving the update tuple, each VSet member v_i measures the RTT between itself and ip (Figure 1(b)), and computes the *error ratio*

$$\delta_{(v_i, g)} = \frac{|RTT(v_i, ip) - \|C - C_{v_i}\||}{RTT(v_i, ip)}$$

where C_{v_i} is v_i ’s coordinate and $\|C - C_{v_i}\|$ is the dis-

tance between the coordinates. Finally, v_i locally stores the *evidence tuple* $(g, \tau, C, ip, \delta_{(v_i, g)})$. Nodes periodically purge tuples that have not recently been queried to reduce storage costs.

4.3 Publisher Coordinate Verification

To update its coordinate, the investigator queries a random node (via a `deliver` message to a random GUID g) in the network (i.e., the publisher). As depicted in Figure 1(c), the publisher replies with a *claim tuple* (g, Γ, τ, C, ip) . The investigator immediately discards the publisher’s coordinates if the publisher’s IP address is not ip , $g \neq H(ip)$, or it deems Γ (VSet size) insufficiently large to offer enough supporting evidence for the coordinate.

Otherwise, the investigator transmits the *evidence query* (g, τ) to each member of the publisher’s VSet, constructed on demand given g according to Eq. 2 (Figure 1(d)). If a VSet member v_i stores an evidence tuple containing both g and τ (logical timestamp), it returns that tuple to the investigator (Figure 1(e)). The investigator then checks that the GUID, network address, and coordinates in the publisher’s claim tuple matches those in the evidence tuple. If there is a discrepancy, the evidence tuple is ignored.

After querying all members of the publisher’s VSet, the investigator counts the number of non-discarded evidence tuples for which $\delta_{(v_i, g)} \leq \hat{\delta}$, where $\hat{\delta}$ is the investigator’s chosen *ratio cutoff parameter*. Intuitively, this parameter gauges the investigator’s tolerance of coordinate errors: a large $\hat{\delta}$ permits fast convergence times when all nodes are honest, but risks increased likelihood of accepting false coordinates. If the count of passing evidence tuples meets or exceeds the investigator’s *evi-*

Dataset	# of Nodes	Pairwise Latency Avg.	Median	System Err. Ratio
Meridian	500	71.3	55.0 ms	0.15
King	500	72.7	63.0 ms	0.09
S ³	359	85.8	67.9 ms	0.17
PL	124	316.4	134.0 ms	0.10

Table 1: Properties of the Meridian, King, S³, and PL pairwise latency datasets, and Vivaldi’s system error ratios for each dataset.

dence cutoff parameter, R , the coordinate is considered verified. Otherwise, the publisher’s coordinate is discarded.

4.4 Tuning VSet Parameters

To determine an appropriate value for the *ratio cutoff parameter* $\hat{\delta}$, we examined Vivaldi’s system error ratio when run against the Meridian [38], King [17], and Scalable Sensing Service (S³) [39] datasets, as well as a pairwise latency experiment that we executed on PlanetLab [24] (PL). Simulations and the PlanetLab experiment used Bamboo [3], a DHT with a Vivaldi implementation and a simulation mode that takes as input a matrix of pairwise latencies. Due to scalability limitations, the simulator used the first 500 nodes from the Meridian and King datasets. Simulation results were averaged over 10 runs. Table 1 provides the properties of the four datasets as well as Vivaldi’s achieved system error ratio.

An appropriate value for $\hat{\delta}$ should be sufficiently large to accommodate baseline errors. For example, in our Veracity implementation (see Section 6.1), we use a ratio cutoff parameter $\hat{\delta}$ of 0.4, well above the system error ratio for all datasets.

In the absence of network churn, the VSet membership of a publisher remains unchanged. With network churn, some of the VSet members may be modified as the keyspace of the directory service is reassigned. New VSet members may not have stored any evidence tuples, but as long as R (*evidence cutoff parameter*) VSet members successfully verify the coordinate, the coordinate can be used. In our experiments, we note that even when R is 4 for a VSet size of 7, Veracity can tolerate moderate to high degrees of churn while ensuring convergence in the coordinate system.

4.5 Candidate Coordinate Verification

The *publisher coordinate verification* scheme described in Section 4.3 provides the investigator with evidence that a publisher’s *coordinate* is accurate. This does not

prevent a malicious publisher from deliberately delaying an investigator’s RTT probe, thereby causing the investigator to update its own coordinate erroneously. (Recall that to update its coordinate, the investigator must measure the RTT between itself and the publisher after having obtained the publisher’s coordinate.)

Once an investigator has validated the publisher’s coordinate, the *candidate coordinate verification* scheme compares coordinate estimation errors among the investigator and a random subset of nodes (the *RSet*) using the investigator’s current coordinate (C_I) and a new candidate coordinate (C'_I) calculated using the publisher’s verified coordinate and the measured RTT.

The investigator queries for the coordinates of Λ RSet members by addressing *deliver* messages to random GUIDs (Figures 2(a) and 2(b)). As with Γ (VSet size), a larger Λ (RSet size) increases confidence in the candidate coordinate at the expense of additional communication. In our experimentation, we found that setting $\Lambda = \Gamma = 7$ provides reasonable security without incurring significant bandwidth overhead. The investigator (I) measures the RTT between itself and each RSet member (Figure 2(c)) and computes the average error ratio

$$err(C, \text{RSet}) = \frac{\left(\sum_{r_j \in \text{RSet}} \frac{|RTT_{Ir_j} - ||C - C_{r_j}|||}{RTT_{Ir_j}} \right)}{\Lambda}$$

for both C_I and C'_I . If the new coordinate causes the error ratio to increase by a factor of more than the *tolerable error factor* Δ , then C'_I is discarded and the investigator’s coordinate remains C_I . Otherwise, the investigator sets C'_I as his new coordinate. The value of Δ must be sufficiently large to permit normal oscillations (e.g., caused by node churn) in the coordinate system. Setting $\Delta \geq 0.2$ enabled Veracity to converge at approximately the same rate as Vivaldi for all tested topologies (we investigate Veracity’s effect on convergence time in Section 6.2.2).

5 Distributed Directory Services

Veracity utilizes DHTs to implement its distributed directory service, which supports the *deliver* messaging functionality described in Section 4.1. While one can adopt a centralized or semi-centralized directory service, a fully-decentralized solution ensures scalability, allowing Veracity’s security mechanisms to be deployable at Internet scale. DHTs are ideal because they scale gracefully with network size, requiring $O(\lg N)$ messages to resolve a GUID [34, 26, 25].

While DHTs ensure scalability, they are vulnerable to insider manipulation [36] due to their distributed nature. Malicious nodes can conduct *Sybil attacks* to increase

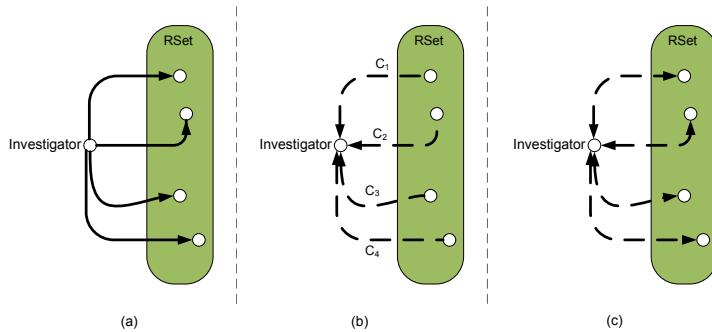


Figure 2: Candidate coordinate verification. Solid lines denote messages sent via `deliver` and dotted lines represent messages sent via direct IP. (a) Investigator queries random nodes (the RSet) for their coordinates. (b) RSet members report their coordinates to Investigator. (c) Investigator measures the RTTs between itself and RSet members, and then calculates the error ratios for the current (C_I) and candidate (C'_I) coordinates.

their influence by registering multiple identities in the network [13], *eclipse attacks* in which they falsify routing update messages to corrupt honest nodes’ routing tables [33], and *routing attacks* in which they inject spurious responses to messages that cross their paths [5]. Fortunately, well-studied techniques exist that defend DHTs against such attacks [11, 10, 4, 14, 5, 1]. We describe defenses that are compatible with Veracity’s design below.

Sybil attack countermeasures that are compatible with a decentralized architecture include *distributed registration* in which *registration nodes*, computed using iterative hashing of a new node’s IP address, vote on whether the new node can join the system based on the number of similar requests it has received from the same IP address [11]. Alternatively, Danezis *et al.* propose using *bootstrap graphs* that capture the relationships between joining nodes and the nodes through which they join to construct trust profiles [10]. Finally, Borisov suggests the use of cryptographic puzzles (e.g., finding a string in which the last p bits of a cryptographic hash are zero) to increase the cost of joining the network [4].

There are also several security techniques that mitigate eclipse and routing attacks. The S-Chord system proposed by Fiat *et al.* organizes the network into *swarms* based on GUIDs [14]. Lookups are relayed between swarms and are forwarded only if the lookup was sent from a majority of the members of the previous swarm. S-Chord is resilient to attacks in which the adversary controls $(1/4 - \epsilon_0)z$ nodes, where $\epsilon_0 > 0$, z is the minimum number of nodes in the network at any time, k is a tunable parameter, and the number of honest nodes is less than or equal to z^k . Castro *et al.* propose the use of *redundant routing* in which queries are sent via diverse paths [5], reaching the intended recipient if all nodes on at least one path are honest. Sanchez *et al.* improve the redundant routing technique in their Cyclone system [1], showing that 85% of requests were correctly delivered when attackers controlled 30% of a 1024 node network and nodes sent messages using 8 redundant paths.

The impact of utilizing the above secure routing techniques is minimal. All of the above approaches operate below the Veracity protocol and do not affect Veracity’s operation. Approaches that rely on redundant messaging incur a linear increase in bandwidth overhead, since all `deliver` messages must be reliably communicated. As we show in Section 6.5.1, Veracity’s communication costs (measured using uncompressed messages) are within the tolerances of even dial-up Internet users. A small linear increase in bandwidth can likely be compensated for by using less expensive message formats (our implementation currently uses Java serialization libraries) and data compression.

We argue that the above DHT security techniques are sufficient to provide the reliability required of Veracity’s `deliver` messaging functionality. Furthermore, unforeseen attacks that manage to circumvent such mechanisms have the effect of artificially increasing the fraction of malicious nodes in the network (since a greater fraction of messages will be misdirected towards misbehaving nodes), and such attacks can be compensated for by increasing R (the number of VSet members that must support a publisher’s claimed coordinate for it to be accepted) and Λ (the RSet size).

Finally, to our best knowledge, Veracity is one of the first attempts at directly addressing the problem of secure distributed directory services and secure neighbor selection in the context of coordinate systems. Existing proposals for securing network coordinate systems either rely on *a priori* trusted nodes [15, 28] or utilize decentralized architectures while ignoring the mechanisms used to locate peers [6, 40], implicitly assuming in the latter case that the underlying coordinate system provides some distributed techniques to securely populate neighbor sets. Unfortunately, such an assumption does not hold as none of the existing systems (Vivaldi [7], PIC [6], ICS [19], the Big-bang simulation [29], nor NPS [22]) describe mechanisms for ensuring that neighbor selection cannot be influenced by mis-

behaving nodes. Veracity utilizes the distributed directory service for both neighbor location and VSet resolution, but other coordinate systems that rely on a directory service solely to determine neighbor sets risk significant vulnerability if the neighbor sets are easily populated by malicious nodes.

6 Implementation and Evaluation

In this section, we evaluate Veracity’s ability to mitigate various forms of attacks in the presence of network churn. We have implemented Veracity by modifying the Vivaldi implementation that is packaged with Bamboo [3], an open-source DHT that is resilient to high levels of node churn [25] and functions either in simulation mode or over an actual network.

6.1 Experimental Setup

Veracity uses Vivaldi as the underlying coordinate system with a 5-dimensional coordinate plane (the recommended configuration in the Bamboo source code [3]). Each node attempted to update its coordinate every 10 seconds. The size of VSets and RSets were both fixed at 7 ($\Gamma = \Lambda = 7$). We used a ratio cutoff parameter $\hat{\delta}$ of 0.4 and an evidence cutoff parameter R of 4. That is, at least 4 of the 7 VSet members had to report error ratios less than 0.4 for a coordinate to be verified. The maximum tolerable increase in error (Δ) for the candidate coordinate verification was set to 0.2.

Our experiments are carried out using Bamboo’s simulation mode as well as on PlanetLab. In the simulation mode, we instantiated 500 nodes with pairwise latencies from the Meridian and King datasets. Due to space constraints, simulation results are shown only for the Meridian dataset. Similar conclusions were drawn from the King dataset and are available in the technical report version of this paper [30]. To distribute the burden of bootstrapping peers, a node joins the simulated network every second until all 500 nodes are present. Nodes join via an already joined peer selected uniformly at random.

In our PlanetLab experiments, the 100 participating nodes joined within 3 minutes of the first node. The selected PlanetLab nodes were chosen in a manner to maximize geographic diversity. The simulation and PlanetLab experiments share a common code base, with the exception of the simulator’s virtualized network layer.

In Sections 6.2 through 6.4, we present our results in simulation mode in the absence and presence of attackers, followed by an evaluation on PlanetLab in Section 6.5. We focus our evaluation on comparing Vivaldi (with no protection scheme) and Veracity based on the accuracy of the coordinate system, convergence time, ability to handle churn, and communication overhead.

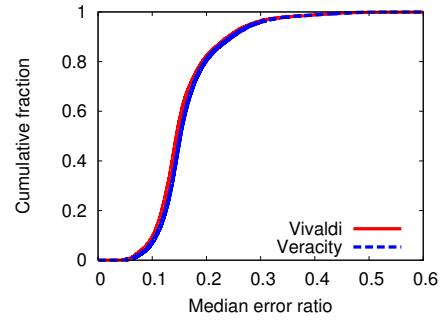


Figure 3: CDFs for median error ratios.

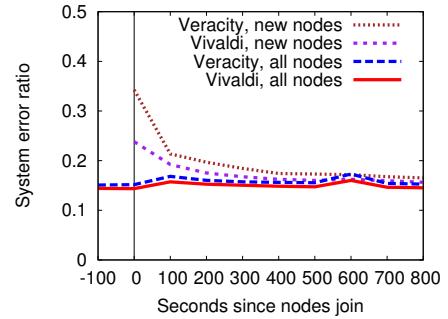


Figure 4: The system error ratio after 10 new nodes join the network (at $t = 0$). The median of the 10 new nodes’ median error ratios is also shown. The coordinate system had stabilized prior to $t = -100$.

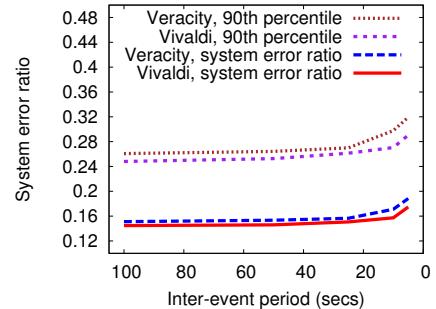


Figure 5: System error ratio for Vivaldi and Veracity under various degrees of churn.

6.2 Veracity in the Absence of Attacks

Before evaluating the effectiveness of Veracity at mitigating various attacks, we first provide a performance comparison between Veracity and Vivaldi in the *absence* of any attackers within the simulation environment.

6.2.1 Accuracy of Network Coordinates

Figure 3 shows the cumulative distribution functions (CDFs) of the median error ratios for Vivaldi and Veracity, computed after the system stabilizes. Veracity raises the system error ratio (the median of the nodes’ median error ratio) by 4.6% (0.79ms) – a negligible difference given latencies over the wide-area. We observe that Veracity and Vivaldi have near identical CDFs, indicating that Veracity’s protection schemes do not significantly influence nodes’ coordinates in the absence of an attack.

6.2.2 Convergence Time

To study how Veracity affects the rate at which the underlying coordinate system converges, we introduce 10 new nodes into the network after the remaining 490 peers have stabilized. Figure 4 plots the system error ratios for Vivaldi and Veracity before and after the new nodes join the network (“all nodes”). The system error ratios of both systems modestly increase when the new nodes are introduced and converge at approximately the same rate. The Figure also shows the median of the 10 new peers’ median error ratios (“new nodes”). Although Veracity incurs a small initial lag in convergence time, the 10 new coordinates quickly reach within 15% of their stabilized (final) value in less than 200 seconds.

The polling frequency – the rate at which nodes attempt to update their coordinate – is directly proportional to the system’s convergence time. Higher polling frequencies enable faster convergence time at the expense of bandwidth. Although the values of the x-axis can be increased or decreased by adjusting the polling frequency, the shape of the curves remain fixed. Repeating our experiments with smaller and larger polling frequencies produced similar results.

6.2.3 Churn Effects

We next compare Vivaldi and Veracity’s ability to handle churn. We adopt the methodology described by Rhea *et al.* [25] for generating churn workloads: a Poisson process schedules events (“node deaths”) in which a node leaves the network. To keep the simulated network fixed at 500 nodes, a fresh node immediately takes the place of a node that leaves. The input to the Poisson process is the expected median inter-event period.

Figure 5 shows the system error ratio for Vivaldi and Veracity for various inter-event periods. Note that the level of churn is inversely proportional to the inter-event period. To illustrate near-worstcase performance, the figure also plots the 90th percentile error ratio.

Both Vivaldi and Veracity are able to tolerate high levels of churn. The “breaking” point of both systems occur

when the inter-event period is less than five seconds, reflecting a rate at which approximately a quarter of the network is replaced every 10 minutes. Churn affects Veracity since the joining and leaving of nodes may cause the members of a VSet to more rapidly change, reducing the investigator’s ability to verify a coordinate. Even at this high churn rate, Veracity’s system error ratio (0.19) is only slightly worse than its error ratio (0.15) when there is no churn. It is worth emphasizing that such high churn (i.e., 25% of the network is replaced every 10 minutes) is unlikely for real-world deployments. The near 0-slope in Figure 5 for inter-event periods greater than 10 seconds shows that neither Vivaldi nor Veracity are significantly affected by more realistic churn rates.

6.3 Disorder Attacks

In this section, we evaluate Veracity’s ability to mitigate *disorder attacks* in which malicious peers report a falsified coordinate chosen at random from a five dimensional hypersphere centered at the origin of the coordinate system. Points are chosen according to Muller’s uniform hypersphere point generation technique [21] with distances from the origin chosen uniformly at random from [0, 2000]. Additionally, attackers delay RTT responses by between 0 and 2000 ms, choosing uniformly at random from that range. Malicious nodes immediately begin their attack upon joining the network.

To emulate realistic network conditions, all simulations experience moderate churn at a median rate of one churn event (a node leaving, immediately followed by a new node joining) every 120 seconds. This churn rate replaces 10% of the nodes during the lifetime of our experiments (100 minutes).

6.3.1 Uncoordinated Attacks

Figure 6 shows the effectiveness of Veracity at mitigating attacks when 10%, 20%, and 30% of peers are malicious. The attackers report a new randomly generated (and false) coordinate whenever probed, randomly delay RTT responses, and are *uncoordinated* (i.e., they do not cooperate). As our baseline, we also include the CDF for Vivaldi in the absence of any attackers.

Malicious attackers significantly reduce Vivaldi’s accuracy, resulting in a 387% increase in the system error ratio (relative to Vivaldi when no attack takes place) even when just 10% of nodes are malicious. When 30% of nodes are malicious, the system error ratio increases dramatically by 1013%. In contrast, Veracity easily mitigates such attacks since the coordinate discrepancies are discernible in evidence tuples, causing inconsistently advertised coordinates to be immediately discarded by investigators. At low rates of attack (10%), the system er-

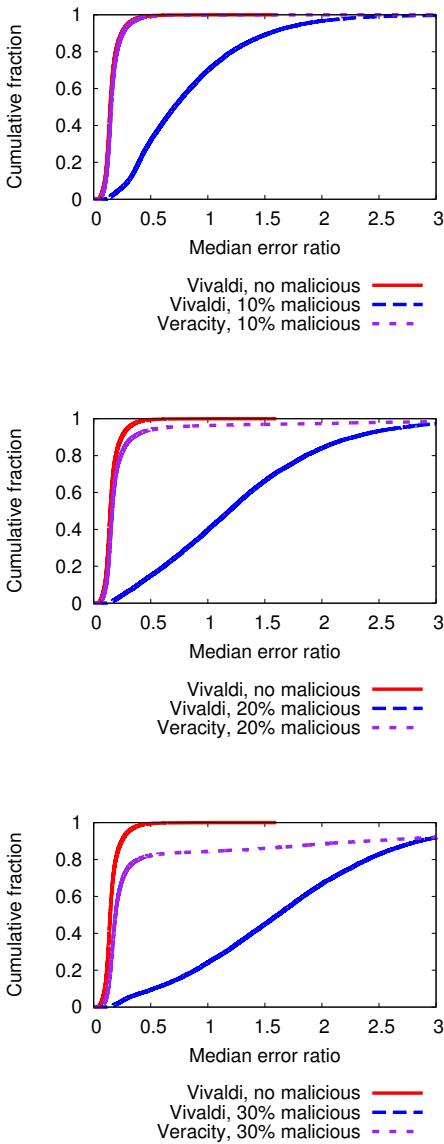


Figure 6: Honest peers’ median error ratios achieved by Vivaldi and Veracity when malicious nodes constitute 10% (*top*), 20% (*middle*), and 30% (*bottom*) of the network. Median error ratios observed when using Vivaldi in a network with no attackers is shown for comparison.

ror ratio increases by only 6% (representing a negligible system-wide median latency error of 4ms). When 30% of the network is malicious, Veracity limits the increase in system error ratio to 32% (5.7ms), an 88% improvement over Vivaldi under the same attack.

Malicious nodes may conduct a more intelligent attack

by randomly delaying probes while reporting *consistent* but erroneous coordinates. That is, each malicious node randomly generates a coordinate and reports the identical (and false) coordinate whenever probed. Such a strategy eliminates coordinate inconsistencies among VSet members. Compared to the previously described attack, this strategy results in lower estimation errors for Vivaldi but does slightly better against Veracity. Here, the increase in Vivaldi’s system error ratio is 163% for 10% malicious nodes and 368% for 30% malicious. Veracity successfully defends against heavy network infiltrations, yielding an increase in the system error ratio of just 39% when 30% of the network is malicious. Veracity reaches its tipping point when 40% of nodes are malicious, incurring an increase of 118%. We note that this increase is still far below the 497% increase experienced by Vivaldi.

6.3.2 Coordinated Attacks

We next consider *coordinated attacks* in which malicious nodes cooperate to increase the effectiveness of their attack. Malicious nodes offer supportive evidence for coordinates advertised by other dishonest nodes and do not offer any evidence for honest peers. That is, when queried, they provide evidence tuples with low (passing) error ratios for malicious nodes and do not respond to requests when the publisher is honest. We conservatively model an attack in which all malicious nodes belong to the same attack coalition. To further maximize their attack, each malicious node randomly generates a fixed erroneous coordinate and advertises it for the duration of the experiment. Additionally, attackers randomly delay RTT responses.

Figure 7 shows Veracity’s performance (measured by the cumulative distribution of median error ratios) when the malicious nodes cooperate. For comparison, the Figure also plots the CDFs for equally sized uncoordinated attacks against Veracity and Vivaldi. Since Vivaldi does not collaborate with peers to asses the truthfulness of advertised coordinates, there is no equivalent “coordinated” attack against Vivaldi.

For all tested attack strengths, the coordinated attacks did not induce significantly more error than uncoordinated attacks. The resultant system error ratios differed little: when attackers control 30% of the network, the system error ratios are 0.202 and 0.201 for the uncoordinated and coordinated attacks, respectively (for comparison, Vivaldi’s system error ratio is 0.679).

6.3.3 Rejected: VSet-only and RSet-only Veracity

The previous sections show that Veracity’s two protections schemes – publisher coordinate verification and candidate coordinate verification – effectively mitigate

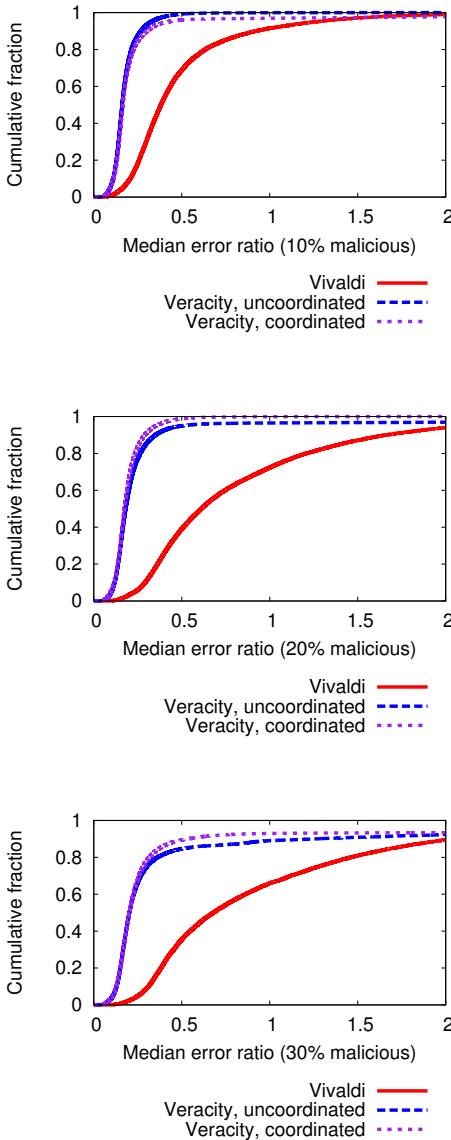


Figure 7: Honest peers’ median error ratios when attackers conduct uncoordinated and coordinated attacks. Attackers comprise 10% (*top*), 20% (*middle*), and 30% (*bottom*) of network peers.

attacks when the adversary controls a large fraction of the network. In this section, we investigate whether it is sufficient to apply only one of the two techniques to achieve similar security.

Figure 8 shows the cumulative distribution of median error ratios when nodes utilize only publisher coordinate verification (“VSet-only”) or candidate coordinate verifi-

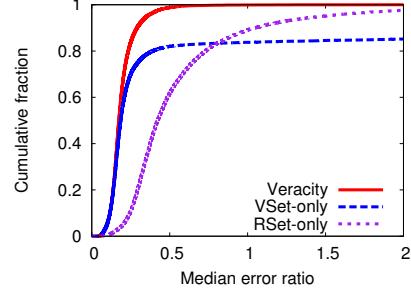


Figure 8: CDF of median error ratios for Veracity, Veracity without Candidate Coordinate Verification (“VSet-only”), and Veracity without Publisher Coordinate Verification (“RSet-only”). The attacker controls 20% of the network and conducts a coordinated attack.

cation (“RSet-only”). We model the attack scenario from Section 6.3.2 in which 20% of the nodes are malicious and cooperating. For comparison, we also show the CDF when both strategies are utilized (“Veracity”). The VSet-only technique achieves nearly the same system error ratio as Veracity (0.19 and 0.17, respectively). However, using only publisher coordinate verification results in a very long tail of median error ratios. In particular, the 90th percentile error ratio is 0.29 for Veracity and 4.42 for VSet-only. Hence, publisher coordinate verification protects the accuracy of most nodes, but permits a significant degradation in accuracy for a minority of peers.

By itself, candidate coordinate verification results in a higher system error ratio (0.42) than VSet-only or Veracity. Additionally, RSet-only has a longer tail than Veracity, resulting in a 90th percentile error ratio of 1.05 during the attack.

By combining both techniques, Veracity better protects the underlying coordinate system, achieving error ratios that nearly mirror those produced by Vivaldi in the absence of attack (see Figures 6 and 7).

6.3.4 Summary of Results

To summarize the performance of Veracity under disorder attacks, Table 2 shows the *relative system error ratio* for various attacker scenarios that we have described, where each system error ratio is normalized by that obtained by Vivaldi under no attacks.

Overall we observe that Veracity is effective at mitigating the effects of disorder attacks. Even under heavy attack (40% malicious nodes), disorder attacks result in a relative system error of 1.54, far below Vivaldi’s relative median error of 13.9.

Percentage of malicious nodes	Inconsistent coords (Uncoordinated)		Consistent coords (Uncoordinated)		Consistent coords (Coordinated)
	Vivaldi	Veracity	Vivaldi	Veracity	Veracity
0%	1.00	1.05	1.00	1.05	1.05
10%	4.87	1.06	2.63	1.11	1.10
20%	8.18	1.12	4.21	1.25	1.22
30%	11.13	1.32	4.68	1.39	1.48
40%	13.90	1.54	5.97	2.18	2.37

Table 2: Relative system error ratios (system error ratio of the tested system divided by the system error ratio of Vivaldi when no attack takes place) for various attacker scenarios.

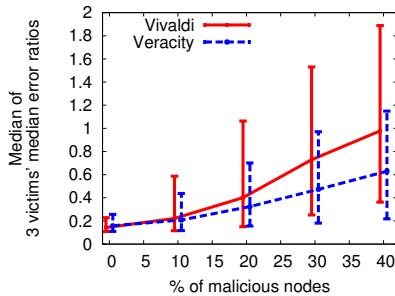


Figure 9: Effects of a combined repulsion and isolation attack against three victim nodes. Points represent the median with error bars denoting the 10th and 90th percentile of the median error ratios of the three victims. For readability, datapoints are slightly shifted along the x-axis by -0.5 for Vivaldi and $+0.5$ for Veracity.

Veracity's effectiveness matches or exceeds that of the prior proposals discussed in Section 7. In contrast to existing coordinate protection systems, Veracity does not require pre-selected trusted nodes, triangle inequality testing, nor outlier detection based on a fixed neighbor set, and is therefore better suited for practical deployment.

6.4 Repulsion and Isolation Attacks

While Veracity is intended primarily to defend against disorder attacks, our next experiment demonstrates the effectiveness of Veracity for protecting against repulsion and isolation attacks. We carry out a combined repulsion and isolation attack as follows: malicious nodes are partitioned into three coalitions, each of which attempts to repulse and isolate a single victim node. Attackers attempt to repulse the targeted node towards an extremely negative coordinate (i.e., having -1000 in all five dimensions) by using the following heuristic: if the victim is

closer than the attacker to the negative coordinate, the attacker behaves honestly. Otherwise, the attacker reports his accurate coordinate but delays the victim investigator's RTT probe response by 1000ms, causing the victim to migrate his coordinate (provided it passes candidate coordinate verification) towards the negative coordinate.

Figure 9 shows the median of the three victim nodes' median error ratios achieved during the combined repulsion and isolation attack. In contrast to previous experiments, we do not use the system error ratio (the median over all peers' median error ratios), as repulsion and isolation attacks target specific victims and need not cause a significant degradation in coordinate accuracy for the remaining peers.

Veracity consistently offers lower median error ratios than Vivaldi. While Veracity does not completely mitigate the effects of repulsion and isolation attacks, our results suggest that the vote-based verification scheme is amenable to defending against such attacks.

6.5 PlanetLab Results

In our last experiment, we validate our simulation results by deploying Veracity on the PlanetLab testbed.

6.5.1 Communication Overhead

To quantitatively measure Veracity's communication overhead in practice, we analyze packet traces recorded on approximately 100 PlanetLab nodes for both Vivaldi and Veracity. Traces are captured using tcpdump and analyzed using the tcptrace network flow analyzer [12]. Figure 10 shows the per-node bandwidth (averaged over all nodes) utilization (KBps) for Vivaldi and Veracity.

Veracity incurs a communication overhead since publishers' coordinates must be verified by VSets and investigators' candidate coordinates must be assessed by RSets. Since Veracity uses a DHT as its directory service, it leverages the scalability of DHTs: each verification step requires $O((\Gamma + \Lambda) \lg N)$, where Γ and Λ denotes the VSet and RSet sizes respectively.

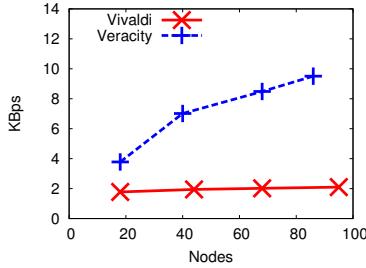


Figure 10: Bandwidth (KBps) on PlanetLab.

Based on the PlanetLab measurements, we performed logarithmic regression analysis to extrapolate the per-node bandwidth requirements of Veracity as the number of nodes increases: $0.1895 \log N + 1.228$ KBps ($r^2 = 0.998$) for Vivaldi and $3.591 \log N - 6.499$ KBps ($r^2 = 0.994$) for Veracity. Figure 11 shows the extrapolated bandwidth utilization of Vivaldi and Veracity for large networks. For a large network consisting of 100,000 nodes, Veracity’s expected per-node bandwidth requirement is a modest 35KBps, making it accessible to typical broadband users.

6.5.2 Accuracy Under Disorder Attacks

Figure 12 plots the system error ratio achieved on PlanetLab for varying attacker infiltrations. Malicious nodes advertise inaccurate (but consistent) coordinates, delay RTT responses, and do not coordinate their attack. To calculate error ratios (which requires knowledge of actual pairwise RTT measurements), we utilize RTT data from our PlanetLab “all-pairs-ping” experiment (see Figure 1). We observe that Veracity effectively mitigates attacks, yielding an increase in system error ratio (relative to Vivaldi under no attack) of just 38% when 32% of the network is malicious. In contrast, Vivaldi suffers an increase of 1679% when 31% of the nodes are dishonest. (The slight differences between attacker percentages is due to the intermittent availability of PlanetLab nodes.)

7 Related Work

Kaafar *et al.* [16] first identified the vulnerability of coordinate systems, in which just 5% of the participating nodes can render the system unusable simply by either lying about its coordinates or delaying RTT probe replies. Subsequently, there have been several recent proposals targeted at securing coordinate systems.

PIC detects dishonest nodes by observing that falsified coordinates or delayed measurements likely induce trian-

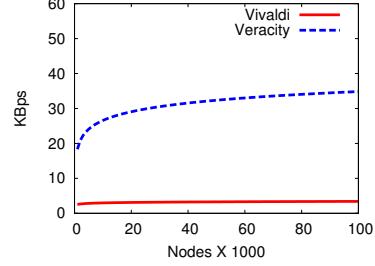


Figure 11: Extrapolated bandwidth (KBps) for large networks.

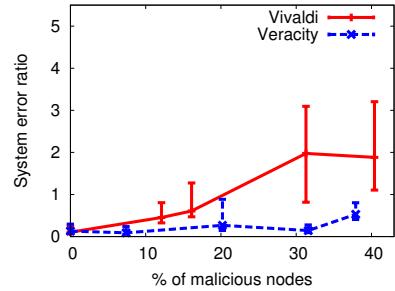


Figure 12: System error ratio achieved on PlanetLab. Error bars denote the 10th and 90th percentile error ratios.

gle inequality violations (TIVs) [6]. To verify peers’ coordinates and measurements, honest nodes use distances to trusted landmarks to detect TIVs. Using a generated transit-stub topology of 2000 nodes, PIC is able to tolerate attacks when up to 20% of the network was controlled by colluding adversaries [6]. However, more recent work has indicated that TIVs can potentially be common and persistent [20], reducing the practicality of PIC’s protection scheme on real-world networks.

Kaafar *et al.* propose the use of trusted *surveyor nodes* to detect malicious behavior [15]. Surveyor nodes position themselves in the coordinate space using only other trusted surveyors. Nodes profile surveyors to model honest behavior, detecting falsified coordinates and measurements as behavior that differs from their constructed model. Kaafar *et al.* conclude that their approach is effective when 30% or less of the network is controlled by malicious and cooperating nodes [15]. Their technique requires 8% of the nodes to be *a priori* trusted surveyors [15] – a nontrivial fraction when the network consists of 100,000 or more nodes.

The RVivaldi system proposed by Saucez *et al.* protect coordinate systems using surveyors as well as centralized *Reputation Computation Agents* (RCAs), the latter of which assigns reputations (trust profiles) to coordinates [28, 27]. Their technique is evaluated only against non-cooperating adversaries, and tolerates up to 20% malicious nodes [28].

Like Veracity, the system proposed by Zage and Nitisoraru is fully distributed and designed for potentially wide-scale deployments [40]. Their approach relies on outlier detection, reducing the influence of nodes whose coordinates are too distant (*spatial locality*) or whose values change too rapidly in short periods of time (*temporal locality*). Their technique successfully mitigates attacks when 30% or fewer of the nodes are under an attacker’s control [40]. However, the temporal locality heuristic requires that each node maintains an immutable *neighborset*, a list of neighbors that a node uses to update its coordinates. Wide-scale deployments involving hundreds of thousands of nodes are likely to be dynamic with nodes frequently joining and leaving the system. The high rate of churn will lessen the opportunities for temporal analysis as nodes leave the system (since less history is available), and cause errors in such analysis for newly joined nodes for which frequent changes in coordinates are expected. In contrast, Veracity does not discriminate against spatial or temporal outliers, and as described in Section 6.2.3, tolerates high levels of churn.

This paper extends our original position paper [32] that outlines our initial design of Veracity. This paper additionally proposes a second verification step geared towards ensuring the correctness of coordinate updates in the presence of malicious delays in latency measurements. This paper further presents a full-fledged implementation that is experimented within a network simulation environment and on PlanetLab.

8 Conclusion

This paper proposes *Veracity*, a fully distributed service for securing network coordinates. We have demonstrated through extensive network simulations on real pairwise latency datasets as well as PlanetLab experiments that Veracity effectively mitigates various forms of attack. For instance, Veracity reduces Vivaldi’s system error ratio by 88% when 30% of the network misbehaves by advertising inconsistent coordinates and adding artificial delay to RTT measurements. Veracity performs well even against cooperating attackers, reducing Vivaldi’s system error ratio by 70% when 30% of the network is corrupt and coordinates its attacks.

We argue that Veracity provides a more practical path to deployment while providing equivalent (or greater) se-

curity than previously proposed coordinate security systems. Unlike PIC, Veracity does not associate triangle-inequality violations (TIVs) with malicious behavior [6], and as indicated by our simulation and PlanetLab results, does not impose additional inaccuracies in the coordinate system when TIVs do exist. Veracity is fully decentralized, requiring no *a priori* shared secrets or trusted nodes. In comparison to techniques that require specialized trusted nodes [28, 27, 15], Veracity is well-suited for applications for which centralized trust models are incompatible (e.g., anonymity networks [31]), and in general, removes central points of trust that may serve as focal points of attack. Veracity’s use of distributed directory services enables graceful scalability, and hence the system can easily be applied to wide-scale virtual coordinate system deployments.

Our most immediate future work entails the use of secure network coordinate systems to permit applications to intelligently form routes that meet specific latency and bandwidth requirements. We are also investigating anonymity services [31] that may leverage Veracity to produce high-performance anonymous paths. Finally, we expect to release an open-source implementation of Veracity in the near future.

Acknowledgments

The authors are grateful to our shepherd, Kenneth Yocum, for his insightful comments and advice. We also thank the anonymous reviewers for their many helpful suggestions. This work is partially supported by NSF Grants CNS-0831376, CNS-0524047, CNS-0627579, and NeTS-0721845.

References

- [1] M. S. Artigas, P. G. Lopez, and A. F. G. Skarmeta. A Novel Methodology for Constructing Secure Multipath Overlays. *IEEE Internet Computing*, 9(6):50–57, 2005.
- [2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, Vol. 46, No. 2, Feb. 2003.
- [3] The Bamboo Distributed Hash Table. <http://bamboo-dht.org/>.
- [4] N. Borisov. Computational Puzzles as Sybil Defenses. In *IEEE International Conference on Peer-to-Peer Computing*, pages 171–176, 2006.
- [5] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *OSDI*, 2002.
- [6] M. Costa, M. Castro, R. Rowstron, and P. Key. PIC: Practical Internet Coordinates for Distance Estimation. In *ICDCS*, 2004.

- [7] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. *SIGCOMM*, 34(4):15–26, 2004.
- [8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *SOSP*, 2001.
- [9] F. Dabek, J. Li, E. Sit, F. Kaashoek, R. Morris, and C. Blake. Designing a DHT for Low Latency and High Throughput. In *NSDI*, 2004.
- [10] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, and R. Anderson. Sybil-Resistant DHT Routing. In *European Symposium On Research In Computer Security*, 2005.
- [11] J. Dinger and H. Hartenstein. Defending the Sybil Attack in P2P Networks: Taxonomy, Challenges, and a Proposal for Self-Registration. In *International Conference on Availability, Reliability and Security*, pages 756–763, 2006.
- [12] D. Dittrich. tcpdstat. <http://staff.washington.edu/dittrich/talks/core02/tools/tools.html>.
- [13] J. R. Douceur. The Sybil Attack. In *First International Workshop on Peer-to-Peer Systems*, March 2002.
- [14] A. Fiat, J. Saia, and M. Young. Making Chord Robust to Byzantine Attacks. In *Proc. of the European Symposium on Algorithms*, 2005.
- [15] M. A. Kaafar, L. Mathy, C. Barakat, K. Salamatian, T. Turletti, and W. Dabbous. Securing Internet Coordinate Embedding Systems. In *ACM SIGCOMM*, August 2007.
- [16] M. A. Kaafar, L. Mathy, T. Turletti, and W. Dabbous. Real Attacks on Virtual Networks: Vivaldi out of Tune. In *SIGCOMM Workshop on Large-Scale Attack Defense*, 2006.
- [17] “King” Data Set. <http://pdos.csail.mit.edu/p2psim/kingdata/>.
- [18] J. T. Ledlie. *A Locality-Aware Approach to Distributed Systems*. PhD thesis, Harvard University, September 2007.
- [19] H. Lim, J. C. Hou, and C.-H. Choi. Constructing Internet Coordinate System Based on Delay Measurement. In *IMC*, 2003.
- [20] E. K. Lua, T. G. Griffin, M. Pias, H. Zheng, and J. Crowcroft. On the Accuracy of Embeddings for Internet Coordinate Systems. In *IMC*, 2005.
- [21] M. E. Muller. A Note on a Method for Generating Points Uniformly on n-dimensional Spheres. *Communications of the ACM*, 2(4):19–20, 1959.
- [22] T. S. E. Ng and H. Zhang. A Network Positioning System for the Internet. In *USENIX Annual Technical Conference*, 2004.
- [23] P. Pietzuch, J. Ledlie, M. Mitzenmacher, and M. Seltzer. Network-Aware Overlays with Network Coordinates. In *Distributed Computing Systems Workshops*, July 2006.
- [24] PlanetLab Global Testbed. <http://www.planet-lab.org/>.
- [25] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *USENIX Technical Conference*, June 2004.
- [26] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware*, pages 329–350, 2001.
- [27] D. Sauzez. Securing Network Coordinate Systems. Master’s thesis, Université Catholique de Louvain, June 2007.
- [28] D. Sauzez, B. Donnet, and O. Bonaventure. A Reputation-Based Approach for Securing Vivaldi Embedding System. In *Dependable and Adaptable Networks and Services*, 2007.
- [29] Y. Shavitt and T. Tanel. Big-bang Simulation for Embedding Network Distances in Euclidean Space. In *IEEE Infocom*, April 2003.
- [30] M. Sherr, M. Blaze, and B. T. Loo. Veracity: A Fully Decentralized Secure Network Coordinate Service. Technical Report TR-CIS-08-28, University of Pennsylvania, August 2008. <http://www.cis.upenn.edu/~msherr/papers/veracity-tr-cis-08-28.pdf>.
- [31] M. Sherr, B. T. Loo, and M. Blaze. Towards Application-Aware Anonymous Routing. In *HotSec*, August 2007.
- [32] M. Sherr, B. T. Loo, and M. Blaze. Veracity: A Fully Decentralized Service for Securing Network Coordinate Systems. In *IPTPS*, February 2008.
- [33] A. Singh, T. W. Ngan, P. Druschel, and D. S. Wallach. Eclipse Attacks on Overlay Networks: Threats and Defenses. In *25th IEEE International Conference on Computer Communications (INFOCOM)*, 2006.
- [34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.
- [35] Vuze BitTorrent Client. <http://azureus.sourceforge.net/>.
- [36] D. S. Wallach. A Survey of Peer-to-Peer Security Issues. *Software Security – Theories and Systems*, 2609:253–258, 2003.
- [37] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *OSDI*, 2002.
- [38] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *SIGCOMM*, 2005.
- [39] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S. Lee. S³: A Scalable Sensing Service for Monitoring Large Networked Systems. In *SIGCOMM Internet Network Management Workshop*, 2006.
- [40] D. Zage and C. Nita-Rotaru. On the Accuracy of Decentralized Virtual Coordinate Systems in Adversarial Networks. In *CCS*, 2007.

Decaf: Moving Device Drivers to a Modern Language

Matthew J. Renzelmann and Michael M. Swift

University of Wisconsin–Madison

{mjr, swift}@cs.wisc.edu

Abstract

Writing code to interact with external devices is inherently difficult, and the added demands of writing device drivers in C for kernel mode compounds the problem. This environment is complex and brittle, leading to increased development costs and, in many cases, unreliable code. Previous solutions to this problem ignore the cost of migrating drivers to a better programming environment and require writing new drivers from scratch or even adopting a new operating system.

We present Decaf Drivers, a system for incrementally converting existing Linux kernel drivers to Java programs in user mode. With support from program-analysis tools, Decaf separates out performance-sensitive code and generates a customized kernel interface that allows the remaining code to be moved to Java. With this interface, a programmer can incrementally convert driver code in C to a Java *decaf driver*. The Decaf Drivers system achieves performance close to native kernel drivers and requires almost no changes to the Linux kernel. Thus, Decaf Drivers enables driver programming to advance into the era of modern programming languages without requiring a complete rewrite of operating systems or drivers.

With five drivers converted to Java, we show that Decaf Drivers can (1) move the majority of a driver’s code out of the kernel, (2) reduce the amount of driver code, (3) detect broken error handling at compile time with exceptions, (4) gracefully evolve as driver and kernel code and data structures change, and (5) perform within one percent of native kernel-only drivers.

1 Introduction

Our research is motivated by three factors. First, writing quality device driver code is difficult, as evidenced by the many books and conferences devoted to the subject. The net result of this difficulty is unreliability and unavailability [45]. Second, device drivers are a critical part of operating systems yet are developed by a broad community. Early versions of Unix had only a handful of drivers, totaling a few kilobytes of code, that were written by a single developer—Dennis Ritchie [39]. In contrast, modern versions of Linux include over 3200 driver versions in the kernel source tree, developed by over 300 people and entailing 3 million lines of code [46]. Similarly, Windows Vista was released with over 30,000

available device drivers [2]. As a result, the difficulty of writing drivers has a wide impact. Third, despite attempts to change how drivers are developed, they continue to be written as they have been: in the kernel and in C. A glance at early Unix source code [39] shows that, despite decades of engineering, driver code in modern versions of Linux bears a striking resemblance to drivers in the original versions of Unix.

Efforts at providing a cross-platform driver interface [25, 38], moving driver code to user mode [14, 21, 25, 26, 33, 49] or into type-safe languages or extensions to C [5, 9, 25, 42, 51] have had niche successes but have not seen widespread adoption. While we cannot be sure of the reason, we speculate that use of unfamiliar programming languages and the lack of a migration path have stifled use of these approaches. Prior efforts at reusing existing driver code relied on C extensions not in widespread use, such as CCured [9]. In contrast, systems using popular languages generally require that drivers be written from scratch to gain any advantage [25, 51]. However, many drivers are written by copying and pasting existing code [15, 28]. Thus, it may still be easier for a driver developer to modify an existing C driver than to write a new driver from scratch, even if the environment is simpler to program.

Decaf Drivers takes a best-effort approach to simplifying driver development by allowing most driver code to be written at user level in languages other than C. Decaf Drivers sidesteps many of the above problems by leaving code that is critical to performance or compatibility in the kernel in C. All other code can move to user level and to another language; we use Java for our implementation, as it has rich tool support for code generation, but the architecture does not depend on any Java features. The Decaf architecture provides common-case performance comparable to kernel-only drivers, but reliability and programmability improve as large amounts of driver code can be written in Java at user level.

The goal of Decaf Drivers is to provide a clear migration path for existing drivers to a modern programming language. User-level code can be written in C initially and converted entirely to Java over time. Developers can also implement new user-level functionality in Java.

We implemented Decaf Drivers in the Linux 2.6.18.1 kernel by extending the Microdrivers infrastructure [19]. Microdrivers provided the mechanisms necessary to con-

vert existing drivers into a user-mode and kernel-mode component. The resulting driver components were written in C, consisted entirely of preprocessed code, and offered no path to evolve the driver over time.

The contributions of our work are threefold. First, Decaf Drivers provides a mechanism for converting the user-mode component of microdrivers to Java through cross-language marshaling of data structures. Second, Decaf supports incremental conversion of driver code from C to Java on a function-by-function basis, which allows a gradual migration away from C. Finally, the resulting driver code can easily be modified as the operating system and supported devices change, through both editing of driver code and modification of the interface between user and kernel driver portions.

We demonstrate this functionality by converting five drivers to decaf drivers, and rewriting either some or all of the user-mode C code in each into Java. We find that converting legacy drivers to Java is straightforward and quick.

We analyze the E1000 gigabit network driver for concrete evidence that Decaf simplifies driver development. We find that using Java exceptions reduced the amount of code and fixed 28 cases of missing error handling. Furthermore, updating the driver to a recent version predominantly required changes to the Java code, not kernel C code. Using standard workloads, we show while decaf drivers are slower to initialize, their steady-state performance is within 1% of native kernel drivers.

In the following section we describe the Decaf architecture. Section 3 provides a discussion of our Decaf Drivers implementation. We evaluate performance in Section 4, following with a case study applying Decaf Drivers to the E1000 driver. We present related work in Section 6, and then conclude.

2 Design of Decaf Drivers

The primary goal of Decaf Drivers is to simplify device driver programming. We contend that the key to dramatically improving driver reliability is to simplify their development, and that requires:

- User-level development in a modern language.
- Near-kernel performance.
- Incremental conversion of existing drivers.
- Support for evolution as driver and kernel data structures and interfaces change.

User-level code removes the restrictions of the kernel environment, and modern languages provide garbage collection, rich data structures, and exception handling. Many of the common bugs in drivers relate to improper memory access, which is solved by type-safe languages; improper synchronization, which can be improved with

language support for mutual exclusion; improper memory management, addressed with garbage collection; and missing or incorrect error handling, which is aided by use of exceptions for reporting errors.

Moving driver code to advanced languages within the kernel achieves some of our goals, but raises other challenges. Support for other languages is not present in operating system kernels, in part because the kernel environment places restrictions on memory access [51]. Notably, most kernels impose strict rules on when memory can be allocated and which memory can be touched at high priority levels [11, 32]. Past efforts to support Java in the Solaris kernel bears this out [36]. In addition, kernel code must deal gracefully with low memory situations, which may not be possible in all languages [6].

The Decaf architecture balances these conflicting requirements by *partitioning* drivers into a small kernel portion that contains performance-critical code and a large, user-level portion that can be written in any language. In support of the latter two goals, Decaf provides tools to support migration of existing code out of the kernel and to generate and re-generate marshaling code to pass data between user mode and the kernel.

2.1 Microdrivers

We base Decaf Drivers on *Microdrivers*, a user-level driver architecture that provides both high performance and compatibility with existing driver and kernel code [19]. Microdrivers partition drivers into a kernel-level *k-driver*, containing only the minimum code required for high-performance and to satisfy OS requirements, and a user-level *u-driver* with everything else. The k-driver contains code with high bandwidth or low-latency requirements, such as the data-handling code and driver code that executes at high priority, such as interrupt handlers. The remaining code, which is often the majority of code in a driver, executes in a user-level process. While the kernel is isolated from faults in the user-level code, systems such as SafeDrive [52] or XFI [16] can be used to isolate and recover from faults in the kernel portion.

To maintain compatibility with existing code, the *DriverSlicer* tool can create microdrivers from existing driver code. This tool identifies high-priority and low-latency code in drivers that must remain in the kernel and creates two output files: one with functions left in the kernel (the k-driver), and one with everything else (the u-driver). With assistance from programmer annotations, DriverSlicer generates RPC-like stubs for communication between the k-driver and u-driver. The kernel invokes microdrivers normally by either calling into the k-driver or into a stub that passes control to the u-driver.

The Microdrivers architecture does not support several features necessary for widespread use. First, after split-

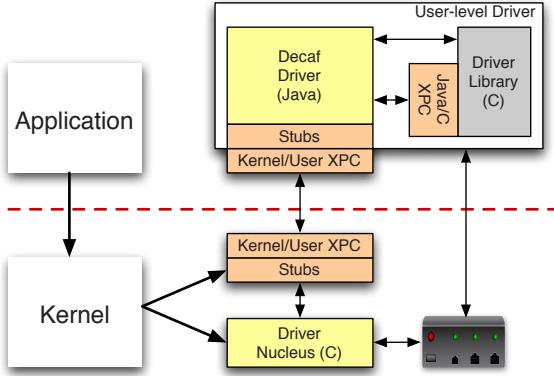


Figure 1: The Decaf Drivers architecture. The OS kernel invokes driver nucleus code or stubs that communicate with the decaf driver via an extension procedure call (XPC).

ting the driver, Microdrivers produces only preprocessed C output, which is unsuitable for evolving the driver once split. Second, Microdrivers only supports C in the u-driver, and provides no facility for moving to any other language.

2.2 Decaf Drivers Overview

Decaf Drivers extends Microdrivers by addressing the deficiencies outlined previously: Decaf Drivers supports (1) writing user-level code in a language other than C, (2) incremental conversion of a legacy driver to a new driver architecture, and (3) evolving the driver as interfaces and data structures change.

Decaf Drivers partitions drivers into two major components: the *driver nucleus*¹ that executes in the kernel for performance and compatibility; and the user-level *decaf driver* written in any language that supports marshaling/unmarshaling of data structures. However, user-level driver code may need to perform actions that are not expressible in all languages, such as directly controlling the hardware with instructions such as *outb*. This code resides in the user-level *driver library*, which executes normal C code. The driver library also provides a staging ground when migrating C code out of the kernel, where it can execute before being converted to another language.

While the architecture supports any language, our implementation is written for Java and we refer to code in the decaf driver as being written in Java. Using Java raises the issue of communicating data structures between languages, in contrast to C++. We believe that other languages that provide mechanisms for invoking native C code, such as Python, would also work well with the Decaf Drivers architecture.

At runtime, all requests to the driver enter through

¹We re-christened the k-driver and u-driver from Microdrivers to more descriptive names reflecting their purpose and implementation, not just their execution mode.

the kernel. The kernel directly invokes functionality implemented by the driver nucleus. Functionality implemented at user level enters through a stub that transfers control to user level and dispatches it to the driver library or the decaf driver. The user-level components may invoke each other or call back into the kernel while processing a request.

The Decaf architecture consists of two major components:

1. *Extension Procedure Call (XPC)* for communication between kernel/user level and between C and Java.

2. *DriverSlicer* to generate marshaling code for XPC. We next discuss these two components in detail.

2.3 Extension Procedure Call

Extension procedure call, created as part of the Nooks driver isolation subsystem [45], provides procedure calls between protection domains. XPC in Decaf Drivers provides five services to enable this cooperation: *control transfer* to provide procedure call semantics (*i.e.*, block and wait); *object transfer* to pass language-level objects, such as structures, between domains; *object sharing* to allow an object to exist in multiple domains; and *synchronization* to ensure consistency when multiple domains access a shared object. *Stubs* invoke XPC services for communication between domains.

The two primary domains participating in driver execution are the driver nucleus and the decaf driver. However, driver functionality may also exist in the driver library, both when migrating code to another language or for functionality reasons. For example, code shared across operating systems may be left in C. Thus, the Decaf architecture also provides XPC between the decaf driver and the driver library to provide access to complex data structures requiring conversion between languages. The decaf driver may directly invoke code in the driver library for simple library calls.

Cross-Domain Control Transfer. The control-transfer mechanism performs the actions of the runtime in an RPC system [4] to pass control from the calling thread to a thread in the target domain. If the decaf driver and the driver library execute in a single process, the control transfer mechanism can be optimized to re-use the calling thread rather than scheduling a new thread to handle the request.

Cross-Domain Object Transfer. XPC provides customized marshaling of data structures to copy only those fields actually accessed at the target. Thus, structures defined for the kernel’s internal use but shared with drivers are passed with only the driver-accessed fields. In addition, XPC provides cross-language conversion, converting structures making heavy use of C language features

for performance (*e.g.*, bit fields) to languages without such control over memory layout.

Object Sharing. Driver components may simultaneously process multiple requests that reference the same object. If two threads are accessing the same object, they should work on a single copy of this object, as they would in the kernel, rather than on two separate copies. Similar to Nooks [45], Decaf Drivers XPC uses an object tracker that records each shared object, extended to support two user-level domains. When transferring objects into a domain, XPC consults the object tracker to find whether the object already exists. If so, the existing object can be updated, and if not, a new object must be allocated.

Synchronization. Synchronized access to data is a challenging problem for regular device drivers because they are reentrant. For example, a device may generate an interrupt while a driver is processing an application request, and the interrupt handler and the request handler may access the same data. To prevent corruption, driver writers must choose from a variety of locking mechanisms based on the priority of the executing code and of potential sharers [29].

The Decaf Drivers synchronization mechanism provides regular locking semantics. If code in one domain locks an object, code in other domains must be prevented from accessing that object while the lock is held. Furthermore, Decaf ensures that the holder of a lock has the most recent version of the objects it protects.

Stubs. Similar to RPC stubs, XPC stubs contain calls specific to a single remote procedure: calls into marshaling code, object tracking code, and control transfer code. These can be written by hand or generated by the DriverSlicer tool. Calls to native functions must be replaced with calls to stubs when the function is implemented in another domain.

2.4 DriverSlicer

The XPC mechanism supports the execution of Decaf Drivers, but does little on its own to simplify the writing of drivers. This task is achieved by the DriverSlicer tool, which enables creation of decaf drivers from existing kernel code written in C. DriverSlicer provides three key functions: (1) partitioning, to identify code that may run outside the kernel, (2) stub generation to enable communication across language and process boundaries, and (3) generation of the driver nucleus and user-level C code to start the porting process. Furthermore, DriverSlicer can regenerate stubs as the set of supported devices, driver data structures, and kernel interfaces change.

Partitioning. Given an existing driver, DriverSlicer automatically partitions the driver into the code that must remain in the kernel for performance or functionality reasons and the code that can move to user level. This fea-

ture is unchanged from the Microdrivers implementation of DriverSlicer. As input, it takes an existing driver and type signatures for *critical root functions*, *i.e.*, functions in the kernel-driver interface that must execute in the kernel for performance or functionality reasons. DriverSlicer outputs the set of functions reachable from critical root functions, all of which must remain in the kernel. The remaining functions can be moved to user level. In addition, DriverSlicer outputs the set of entry-point functions, where control transfers between kernel mode and user mode. The user-mode entry points are the driver interface functions moved to user mode. The kernel entry points are OS kernel functions and critical driver functions called from user mode.

Stub Generation. DriverSlicer creates stubs automatically based on the set of kernel and user entry points output from the partitioning stage. With the guidance of programmer annotations [19], DriverSlicer automatically generates marshaling code for each entry-point function. In addition, DriverSlicer emits code to marshal and unmarshal data structures in both C and Java, allowing complex data structures to be accessed natively in both languages.

Driver Generation. DriverSlicer emits C source code for the driver nucleus and the driver library. The driver library code can be ignored when functions are rewritten in another language. The source code produced is a partitioning of the original driver source code into two source trees. Files in each tree contain the same include files and definitions, but each function is in only one of the versions, according to where it executes.

To support driver evolution, DriverSlicer can be invoked repeatedly to generate new marshaling code as data structures change. The generated driver files need only be produced once since the marshaling code is segregated from the rest of the driver code.

2.5 Summary

The Decaf architecture achieves our four requirements. The decaf driver itself may be implemented in any language and runs at user level. The driver nucleus provides performance near that of native kernel drivers. DriverSlicer provides incremental conversion to C through automatic generation of stubs and marshaling code both for kernel-user communication and C-Java communication. Finally, DriverSlicer supports driver evolution through regeneration of stubs and marshaling code as the driver changes.

3 Implementation

We implemented the Decaf Drivers architecture for the Linux 2.6.18.1 kernel and re-wrote five device drivers into decaf drivers. We use a modified version of DriverSlicer from Microdrivers [19] to generate code for XPC

stubs and marshaling, and implemented extensions to generate similar code between the driver library and decaf driver.

The driver nucleus is a standard Linux kernel module and the decaf driver and driver library execute together as a multithreaded Java application. Our implementation relies on Jeannie [22] to simplify calling from C into Java and back. Jeannie is a compiler that allows mixing Java and C code at the expression level, which simplifies communication between the two languages. Languages with native support for cross-language calls, such as C# [34], provide the ability to call functions in different languages, but do not allow mixing expressions in different languages.

Decaf Drivers provides runtime support common to all decaf drivers. The runtime for user-level code, the *decaf runtime*, contains code supporting all decaf drivers. The kernel runtime is a separate kernel module, called the *nuclear runtime*, that is linked to every driver nucleus. These runtime components support synchronization, object sharing, and control transfer.

3.1 Extension Procedure Call

Decaf Drivers uses two versions of XPC: one between the driver nucleus and the driver library, for crossing the kernel boundary; and another between the driver library and the decaf driver, for crossing the C-Java language boundary. XPC between kernel and user mode is substantially similar to that in Microdrivers, so we focus our discussion on communication between C and Java.

The Decaf implementation always performs XPCs to and from the kernel in C, which allows us to leverage existing stub and marshaling support from Microdrivers. An upcall from the kernel always invokes C code first, which may then invoke Java code. Similarly, downcalls always invoke C code first before invoking the kernel. While this adds extra steps when invoking code in the decaf driver, it adds little runtime overhead as shown by the experiments in Section 4.

3.1.1 Java-C Control and Data Transfer

Decaf Drivers provides two mechanisms for the decaf driver to invoke code in the driver library: direct cross-language function calls and calls via XPC. Direct calls may be used when arguments are scalar values that can be trivially converted between languages, such as arguments to low-level I/O routines. XPC must be used when arguments contain pointers or complex data structures to provide cross-language translation of data types. In addition, downcalls from the decaf driver to the driver nucleus require XPC.

In both cases, Decaf Drivers relies on the Jeannie language [22] to perform the low-level transfer between C and Java. Jeannie enables C and Java code to be mixed in

a source file at the granularity of a single expression. The backtick operator (`) switches between languages. From a combined source file, the Jeannie compiler produces a C file, a Java file, and Java Native Interface code allowing one to call the other. Jeannie provides a clean syntax for invoking a Java function from C and vice versa. When invoking simple functionality in C, the decaf driver can inline the C code right into a Java function.

When invoking a function through XPC, Decaf Drivers uses RPC-style marshaling to transfer complex objects between Java and C. While Jeannie allows code in one language to read variables declared in the other, it does not allow modifications of those variables. Instead, Decaf uses the XDR marshaling standard [13] to marshal data between the driver library and the decaf driver, which we discuss in Section 3.2.3.

We write Decaf stubs in Jeannie to allow pure Java code to invoke native C code. The stubs invoke XDR marshaling code and the object tracker. Figure 2 shows an example of a stub in Jeannie. As shown in this figure, the following steps take place when calling from the decaf driver to the driver nucleus:

1. The decaf driver calls the Jeannie stub.
2. The stub invokes the object tracker to translate any parameters to their equivalent C pointers.
3. The stub, acting as an XPC client, invokes an XDR routine to marshal the Java parameters.
4. While marshaling these parameters, the XDR code uses inheritance to execute the appropriate marshaling routine for the object.
5. The same stub then acts as an XPC server, and unmarshals the Java objects into C.
6. While unmarshaling return values, the C stubs call specialized functions for each type.

We write stubs by hand because our code generation tools can only produce pure Java or C code, but the process could be fully automated.

3.1.2 Java-C Object Sharing

Object sharing maintains the relationship between data structures in different domains with an *object tracker*. This service logically stores mappings between C pointers in the driver library, and Java objects in the decaf driver. Marshaling code records the caller’s local addresses for objects when marshaling data. Unmarshaling code checks the object tracker before unmarshaling each object. If found, the code updates the existing object with its new contents. If not found, the unmarshaling code allocates a new object and adds an association to the object tracker. For kernel/user XPC, the unmarshaling code in the kernel consults the object tracker with a simple procedure call, while unmarshaling code in the driver library

```

Class Ens1371 {
    ...
    public static int snd_card_register(snd_card java_card) {
        CPointer c_card = JavaOT.xlate_j_to_c (java_card); ← Consult object tracker
        int java_ret;
        begin_marshaling ();
        copy_XDR_j2c (java_card); ← Marshal arguments
        end_marshaling ();
        java_ret = `snd_card_register ((void *) `c_card.get_c_ptr()); ← Call C function
        begin_marshaling ();
        java_card = (snd_card) copy_XDR_c2j (java_card, c_card); ← Marshal out parameters
        end_marshaling ();
        return java_ret;
    }
}

```

Figure 2: Sample Jeannie stub code for calling from Java to C. The backtick operator ` switches the language for the following expression, and is needed only to invoke the C function.

must call into the kernel.

The different data representations in C and Java raise two difficulties. First, Java objects do not have unique identifiers, such as the address of a structure in C. Thus, the decaf runtime uses a separate user-level object tracker written in Java, which uses object references to identify Java objects. C objects are identified by their address, cast to an integer.

Second, a single C pointer may be associated with multiple Java objects. When a C structure contains another structure as its first member, both inner and outer structures have the same address. In Java, however, these objects are distinct. This difference becomes a problem when a decaf driver passes the inner structure as an argument to a function in the driver library or driver nucleus. The user-level object tracker disambiguates the uses of a C pointer by additionally storing a *type identifier* with each C pointer.

When an object is initially copied from C to Java, marshaling code adds entries to the object tracker for its embedded structures. When an embedded structure is passed back from Java to C, the marshaling code will search for a C pointer with the correct type identifier. The object tracker uses the address of the C XDR marshaling function for a structure as its identifier.

Once an object's reference is removed from the object tracker, Java's garbage collection can free it normally. We have not yet implemented automatic collection of shared objects, so decaf drivers must currently free shared objects explicitly. Implementing the object tracker with weak references [20] and finalizers would allow unreferenced objects to be removed from the object tracker automatically.

3.1.3 Synchronization

Decaf Drivers relies on kernel-mode *combolocks* from Microdrivers to synchronize access to shared data across domains [19]. When acquired only in the kernel, a combolock is a spinlock. When acquired from user mode, a combolock is a semaphore, and subsequent kernel

threads must wait for the semaphore. Combolocks also provide support for multiple threads in the decaf driver and allow these threads to share data with the driver nucleus and driver library.

However, combolocks alone do not completely address the problem. The driver nucleus must not invoke the decaf driver while executing high priority code or holding a spinlock. We use three techniques to prevent high-priority code from invoking user-level code. First, we direct the driver to avoid interrupting itself: the nuclear runtime disables interrupts from the driver's device with `disable_irq` while the decaf driver runs. Since user-mode code runs infrequently, we have not observed any performance or functional impact from deferring code.

Second, we modify the kernel in some places to not invoke the driver with spinlocks held. For example, we modified the kernel sound libraries to use mutexes, which allowed more code to execute in user mode. In its original implementation, the sound library would often acquire a spinlock before calling the driver. Driver functions called with a spinlock held would have to remain in the kernel because invoking the decaf driver would require invoking the scheduler. In contrast, mutexes allow blocking operations while they are held, so we were able to move additional driver functions into the decaf driver.

Third, we deferred some driver functionality to a worker thread. For example, the E1000 driver uses a watchdog timer that executes every two seconds. Since the kernel runs timers at high priority, it cannot call up to the decaf driver when the timer fires. Instead, we convert timers to enqueue a work item, which executes on a separate thread and allows blocking operations. Thus, the watchdog timer can execute in the decaf driver.

3.2 DriverSlicer Implementation

DriverSlicer automates much of the work of creating decaf drivers. The tool is a combination of OCaml code written for CIL [35] to perform static analysis and generate C code, Python scripts to post-process the generated

C code, and XDR compilers to produce cross-language marshaling code. DriverSlicer takes as input a legacy driver with annotations to specify how C pointers and arrays should be marshaled and emits stubs, marshaling routines, and separate user and kernel driver source code files.

3.2.1 Generating Readable Code

A key goal of Decaf Drivers is support for continued modification to drivers. A major problem with DriverSlicer from microdrivers is that it only generated preprocessed driver code, which is difficult to modify. The Decaf DriverSlicer instead patches the original source, preserving comments and code structure. It produces two sets of files; one set for the driver nucleus and one set for the driver library, to be ported to the decaf driver. This patching process consists of three steps.

First, scripts parse the preprocessed CIL output to extract the generated code (as compared to the original driver source). This code includes marshaling stubs and calls to initialize the object tracker. Other preprocessed output, such as driver function implementations, are ignored, as this code will be taken from the original driver source files instead.

Second, DriverSlicer creates two copies (user and kernel) of the original driver source. From these copies, the tool removes function implemented by the other copy. Any functions in the driver nucleus source tree that are now implemented in the driver library and any functions in the driver library source tree that are implemented in the driver nucleus or the kernel are either replaced with stubs or removed entirely. The stubs containing marshaling code are placed in a separate file to preserve the readability of the patched driver.

Finally, DriverSlicer makes several other minor modifications to the output. It adds `#include` directives to provide definitions for the functions used in the marshaling code, and adds a function call in the driver nucleus `init_module` function to provide additional initialization.

3.2.2 Generating XDR Interface Specifications

The decaf driver relies on XDR marshaling to access kernel data structures. DriverSlicer generates an XDR specification for the data types used in user-level code from the original driver and kernel header files. The existing annotations needed for generating kernel marshaling code are sufficient to emit XDR specifications.

Unfortunately, XDR is not C and does not support all C data structures, specifically strings and arrays. DriverSlicer takes additional steps to convert C data types to compatible XDR types with the same memory layout. First, DriverSlicer discards most of the original code except for `typedefs` and structure definitions. Driver-

Original Structure:

```
struct e1000_adapter {
    ...
    struct e1000_tx_ring test_tx_ring;
    struct e1000_rx_ring test_rx_ring;
    uint32_t * __attribute__((exp(PCI_LEN)))
        config_space;
    int msg_enable;
    ...
};
```

XDR input:

```
struct array256_uint32_t {
    uint32_t array[256];
};

typedef struct array256_uint32_t
*array256_uint32_ptr;

struct e1000_adapter_autoxdr_c {
    ...
    struct e1000_tx_ring test_tx_ring ;
    struct e1000_rx_ring test_rx_ring ;
    array256_uint32_ptr config_space ;
    int msg_enable ;
    ...
};
```

Figure 3: Portions of a driver data structure above, and the generated XDR input below. The names have been shortened for readability. The annotation in the original version is required for DriverSlicer to generate marshaling code between kernel and user levels.

Slicer then rewrites these definitions to avoid functionality that XDR does not support. For example, a driver data structure may include a pointer to a fixed length array. DriverSlicer cannot output the original C definition because XDR would interpret it as a pointer to a single element. Instead, DriverSlicer generates a new structure definition containing a fixed length array of the appropriate type, and then substitutes pointers to the old type with a pointer to the new structure type.

As shown in Figure 3, DriverSlicer converts pointers to an array into a pointer to a structure, allowing XDR to produce marshaling code. This transformation does not affect the in-memory layout. In this way, the generated marshaling code will properly marshal the entire contents of the array. After generating the C output, a script runs which makes a few syntactic transformations, such as converting C's `long long` type to XDR's `hyper` type. The result is a valid XDR specification.

3.2.3 Generating XDR Marshaling Routines

DriverSlicer incorporates modified versions of the `rpcgen` [43] and `jrpcgen` [1] XDR interface compilers to generate C and Java marshaling code respectively. These modifications to the original tools support object

tracking and recursive data structures.

As previously mentioned in Section 3.1.2, the tools emit calls into the object tracker to locate existing versions of objects passed as parameters. The generated unmarshaling code consults the object tracker before allocating memory for a structure. If one is found, the existing structure is used.

The DriverSlicer XDR compilers support recursive data structures, such as circular linked lists. The marshaling code checks each object against a list of the objects that have already been marshaled. When the tool encounters an object again, it inserts a reference to the existing copy instead of marshaling the structure again. This feature extends also across all parameters to a function, so that passing two structures that both reference a third results in marshaling the third structure just once.

The output of DriverSlicer is a set of functions that marshal or unmarshal each data structure used by the functions in the interface. It also emits a Java class for each C data type used by the driver. These classes are containers of public fields for every element of the original C structures. The generated classes provide a useful starting point for writing driver code in Java, but do not take advantage of Java language features. For example, all member variables are public. We expect developers to rewrite these classes when doing more development in Java.

3.2.4 Regenerating Stubs and Marshaling Code

As drivers evolve, the functions implemented in the driver nucleus or the data types passed between the driver nucleus and the decaf driver may change. Consequently, the stubs and marshaling code may need to be updated to reflect new data structures or changed use of existing data structures. While this could be performed manually, DriverSlicer provides automated support for regenerating stubs and marshaling code. Simply re-running DriverSlicer may not produce correct marshaling code for added fields unless it observes code in the user-level partition accessing that field. If this is Java code, it is not visible to CIL, which only processes C code.

When the decaf driver requires access to fields not previously referenced, whether they are new or not, a programmer must inform DriverSlicer to produce marshaling code. DriverSlicer supports an annotation to the original driver code to indicate that a field may be referenced by the decaf driver. A programmer adds the annotation `DECAF_XVAR (y);` where `X` is an R, W, or RW depending on whether the Java code will read, write, or read and write the variable, and `y` is the variable name. These annotations must be placed in entry-point functions through which new fields are referenced.

Thus, the new annotations ensure that DriverSlicer generates marshaling code to allow reading and/or writ-

Source Components	# Lines
Runtime support	
Jeannie helpers	1,976
XPC in Decaf runtime	2,673
XPC in Nuclear runtime	4,661
DriverSlicer	
CIL OCaml	12,465
Python scripts	1,276
XDR compilers	372
<i>Total number of lines of code</i>	<i>23,423</i>

Table 1: The number of non-comment lines of code in the Decaf runtime and DriverSlicer tools. For the XDR compilers, the number of additional lines of code is shown.

ing the new variables in the decaf driver. A programmer can add new functions to the user/kernel interface with similar annotations. In the future, we plan to automatically analyze the decaf driver source code to detect and marshal these fields. In addition, we plan to produce a concise specification of the entry points for regenerating marshaling code, rather than relying on the original driver source.

3.3 Code Size

Table 1 shows the size of the Decaf Drivers implementation. The runtime code, consisting of user-level helper functions written in Jeannie and XPC code in user and kernel mode, totals 9,310 lines. This code, shared by all decaf drivers, is comparable to a moderately sized driver.

DriverSlicer consists of OCaml code for CIL, Python scripts for processing the output, and XDR compilers. As the XDR compilers are existing tools, we report the amount of code we added. In total, DriverSlicer comprises 14,113 lines.

4 Experimental Results

The value of Decaf Drivers lies in simplified programming. The cost of using Decaf Drivers comes from the additional complexity of partitioning driver code and the performance cost of communicating across domain boundaries. We have converted four types of drivers using Decaf Drivers, and report on the experience and the performance of the resulting drivers here. We give statistics for the code we have produced, and answer three questions about Decaf Drivers: how hard is it to move driver code to Java, how much driver code can be moved to Java, and what is the performance cost of Decaf Drivers?

We experimented with the five drivers listed in Table 2. Starting with existing drivers from the CentOS 4.2 Linux distribution (compatible with RedHat Enterprise Linux 4.2) with the 2.6.18.1 kernel, and we converted them to Decaf Drivers using DriverSlicer. All our experiments

except those for the E1000 driver are run on a 3.0GHz Pentium D with 1GB of RAM. The E1000 experiments are run on a 2.5GHz Core 2 Quad with 4GB of RAM. We used separate machines because the test devices were not all available on either machine individually.

4.1 Conversion to Java

Table 2 shows for each driver, how many lines of code required annotations and how many functions were in the driver nucleus, driver library, and decaf driver. After splitting code with DriverSlicer, we converted to Java all the functions in user level that we observed being called. Many of the remaining functions are specific to other devices served by the same driver. The column “Lines of Code” reports the quantity of code in the original driver. The final column, “Orig. LoC” gives the amount of C code converted to Java.

The annotations affect less than 2% of the driver source on average. These results are lower than for Microdrivers because of improvements we made to DriverSlicer to more thoroughly analyze driver code. In addition to the annotations in individual drivers, we annotated 25 lines in common kernel headers that were shared by multiple drivers. These results indicate that annotating driver code is not a major burden when converting drivers to Java. We also changed six lines of code in the `8139too` and `uhci-hcd` driver nuclei to defer functions executed at high priority to a worker thread while the decaf driver or driver library execute; the code is otherwise the same as that produced by DriverSlicer.

While we converted the `8139too` and `ens1371` to Java during the process of developing Decaf Drivers, we converted the other two drivers after its design was complete. For `uhci-hcd`, a driver previously converted to a microdriver, the additional conversion of the user-mode code to a decaf driver took approximately three days. The entire conversion of the `psmouse` driver, including both annotation and conversion of its major routines to Java, took roughly two days. This experience confirms our goal that porting legacy driver code to Java, when provided with appropriate infrastructure support, is straightforward.

In four of the five drivers, we were able to move more than 75% of the functions into user mode. However, we were only able to convert 4% of the functions in `uhci-hcd` to Java because the driver contained several functions on the data path that could potentially call nearly any code in the driver. We expect that redesigning the driver would allow additional code to move to user level. In the `psmouse` driver, we found that most of the user-level code was device-specific. Consequently, we implemented in Java only those functions that were actually called for our mouse device.

The majority of the code that we converted from C to

Java is initialization, shutdown, and power management code. This is ideal code to move, as it executes rarely yet contains complicated logic that is error prone [40].

4.2 Performance of Decaf Drivers

The Decaf architecture seeks to minimize the performance impact of user-level code by leaving critical path code in the kernel. In steady-state behavior, the decaf driver should never or only rarely be accessed. However, during initialization and shutdown, the decaf driver executes frequently. We therefore measure both the latency to load and initialize the driver and its performance on a common workload.

We measure driver performance with workloads appropriate to each type of driver. We use netperf [12] sending and receiving TCP/IP data to evaluate the `8139too` and `E1000` network drivers with the default send and receive buffer sizes of 85 KB and 16 KB. We measure the `ens1371` sound driver by playing a 256Kbps MP3 file. The `uhci-hcd` driver controls low-bandwidth USB 1.0 devices and requires few CPU resources. We measure its performance by untaring a large archive onto a portable USB flash drive and record the CPU utilization. We do not measure the performance of the mouse driver, as its bandwidth is too low to be measurable, and we measure its CPU utilization while continuously moving the mouse for 30 seconds. For all workloads except netperf, we repeated the experiment three times. We executed the netperf workload for a single 600-second iteration because it performs multiple tests internally.

We measure the initialization time for drivers by measuring the latency to run the `insmod` module loader. While some drivers perform additional startup activities after module initialization completes, we found that this measurement provides an accurate representation of the difference between native and Decaf Drivers implementations.

Table 3 shows the results of our experiments. As expected, performance and CPU utilization across the benchmarks was unchanged. With `E1000`, we also tested UDP send/receive performance with 1 byte messages. The throughput is the same as the native driver and CPU utilization is slightly higher.

To understand this performance, we recorded how often the decaf driver is invoked during these workloads. In the `ens1371` driver, the decaf driver was called 15 times, all during playback start and end. A watchdog timer in the decaf driver executes every two seconds in the `E1000` driver. The other workloads did not invoke the decaf driver at all during testing. Thus, the added cost of communication with Java has no impact on application performance.

However, the latency to initialize the driver was sub-

Driver Name	Type	Lines of code	DriverSlicer Annotations	Driver nucleus		Driver library		Decaf driver		
				Funcs	LoC	Funcs	LoC	Funcs	LoC	Orig. LoC
8139too	Network	1,916	17	12	389	16	292	25	541	570
E1000	Network	14,204	64	46	1715	0	0	236	7804	8693
ens1371	Sound	2,165	18	6	140	0	0	59	1049	1068
uhci-hcd	USB 1.0	2,339	94	68	1537	12	287	3	188	168
psmouse	Mouse	2,448	17	15	501	74	1310	14	192	250

Table 2: The drivers converted to the Decaf architecture, and the size of the resulting driver components.

Driver Name	Workload	Relative Performance	CPU Utilization		Init. Latency		User/Kernel Crossings
			native	Decaf	native	Decaf	
8139too	netperf-send	1.00	14 %	13 %	0.02 sec.	1.02 sec.	40
	netperf-recv	1.00	17 %	15 %	–	–	–
E1000	netperf-send	0.99	2.8 %	3.7 %	0.42 sec.	4.87 sec.	91
	netperf-recv	1.00	20 %	21 %	–	–	–
ens1371	mpg123	–	0.0 %	0.1 %	1.12 sec.	6.34 sec.	237
uhci-hcd	tar	1.03	0.1 %	0.1 %	1.32 sec.	2.67 sec	49
psmouse	move-and-click	–	0.1 %	0.1 %	0.04 sec.	0.40 sec.	24

Table 3: The performance of Decaf Drivers on common workloads and driver initialization.

stantially higher, averaging 3 seconds. The increase stems from cross-domain communication and marshaling driver data structures. Table 3 includes the number of call/return trips between the driver nucleus and the decaf driver during initialization. We expect that optimizing our marshaling interface to transfer data directly between the driver nucleus and the decaf driver, rather than unmarshaling at user-level in C and re-marshaling in Java, would significantly reduce the cost of invoking decaf driver code.

5 Case Study: E1000 Network Driver

To evaluate the software engineering benefits of developing drivers in Java, we analyze the Intel E1000 gigabit Ethernet decaf driver. We selected this driver because:

- it is one of the largest network drivers with over 14,200 lines of code.
- it supports 50 different chipsets.
- it is actively developed, with 340 revisions between the 2.6.18.1 and 2.6.27 kernels.
- it has high performance requirements that emphasize the overhead of Decaf Drivers.

With this case study, we address three questions:

1. What are the benefits of writing driver code in Java?
2. How hard is it to update driver code split between the driver nucleus and the decaf driver?
3. How difficult is it to mix C and Java in a driver?

We address these questions by converting the E1000 driver from the Linux 2.6.18.1 kernel to a decaf driver. Overall, we converted 236 functions to Java in the decaf driver and left 46 functions in the driver nucleus. There are no E1000-specific functions in the driver library. Of

the 46 kernel functions, 42 are there for performance or functionality reasons. For example, many are called from interrupt handlers or with a spinlock held.

The four remaining functions are left in the driver nucleus because of an explicit data race that our implementation does not handle. These functions, in the `ethtool` interface, wait for an interrupt to fire and change a variable. However, the interrupt handler changes the variable in the driver nucleus; the copy of the variable in the decaf driver remains unchanged, and hence the function waits forever. This could be addressed with an explicit call into the driver nucleus to wait for the interrupt.

5.1 Benefits from Java

We identified three concrete benefits of moving E1000 code to Java, and one potential benefit we plan to explore.

Error handling. We found the biggest benefit of moving driver code to Java was improved error handling. The standard practice to handle errors in Linux device drivers is through `goto` statements to a set of labels based on when the failure occurred. In this idiom, an `if` statement checks a return value, and jumps to a label near the end of the function on error. The labels are placed such that only the necessary subset of cleanup operations are performed. This system is brittle because the developer can easily jump to an incorrect label or forget to test an error condition [44].

Using exceptions to signal errors and nested handlers to catch errors, however, ensures that no error conditions are ignored, and that cleanup operations take place in the proper order. We rewrote 92 functions to use *checked exceptions* instead of integer return codes. The compiler requires the program to handle these exceptions. In this process, we found 28 cases in which error codes were ig-

```

Decaf driver code:
public static void e1000_open(net_device netdev)
    throws E1000HWEException {
    e1000_adapteradapter = netdev.priv;
    int err;
    try {
        /* allocate transmit descriptors */
        e1000_setup_all_tx_resources(adapter);

        try {
            /* allocate receive descriptors */
            e1000_setup_all_rx_resources(adapter);

            try {
                e1000_request_irq(adapter);
                e1000_power_up_phy(adapter);
                e1000_up(adapter);
                ...
            } catch (E1000HWEException e) {
                e1000_free_all_rx_resources(adapter);
                throw e;
            }
            } catch (E1000HWEException e) {
                e1000_free_all_tx_resources(adapter);
                throw e;
            }
            } catch (E1000HWEException e) {
                e1000_reset(adapter);
                throw e;
            }
    }
}

```

Figure 4: Code converted to nested exception handling.

nored or handled incorrectly. Some, but not all, of these have been fixed in recent Linux kernels.

Figure 4 shows an example from the `e1000_open` function. This code catches and re-throws the exceptions; using a `finally` block would either incorrectly free the resources under all circumstances, or require additional code to ensure the resources are freed only in the face of an error.

Checked exceptions also reduce the amount of code in the driver. Figure 5 shows an example. By switching to exceptions instead of integer return values, we cut 675 lines of code, or approximately 8%, from `e1000_hw.c` by removing code to check for an error and return. We anticipate that converting the entire driver to use exceptions would eliminate more of these checks.

Object orientation. We found benefits from object orientation in two portions of the E1000 driver. In `e1000_param.c`, functions verify module parameters using range and set-membership tests. We use three classes to process parameters during module initialization. A base class provides basic parameter checking, and the two derived classes provide additional functionality. The appropriate class checks each module parameter automatically. The resulting code is shorter than the original C code and more maintainable, because the programmer is forced by the type system to provide ranges and sets when necessary.

In addition, we restructured the hardware accessor functions as a class. In the original E1000 driver,

```

Original Code:
if(hw->ffe_config_state == e1000_ffe_config_active) {
    ret_val = e1000_read_phy_reg(hw, 0x2F5B,
                                  &phy_saved_data);
    if(ret_val) return ret_val;

    ret_val = e1000_write_phy_reg(hw, 0x2F5B, 0x0003);
    if(ret_val) return ret_val;

    msec_delay_irq(20);
    ret_val = e1000_write_phy_reg(hw, 0x0000,
                                  IGP01E1000_IEEE_FORCE_GIGA);
    if(ret_val) return ret_val;
}

```

```

Decaf driver Code:
if(hw.ffe_config_state.value == e1000_ffe_config_active) {
    e1000_read_phy_reg(0x2F5B, phy_saved_data);
    e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
    e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
    DriverWrappers.Java_msleep (20);
    e1000_write_phy_reg((short) 0x0000,
                        (short) IGP01E1000_IEEE_FORCE_GIGA);
}

```

Figure 5: Code from `e1000_config_dsp_after_link_change` in `e1000_hw.c`. The upper box shows the original code with error handling code. The lower box shows the same code rewritten to use exceptions.

these functions all required a parameter pointing to an `e1000_hw` structure. Just rewriting this code as a class removed 6.5KB of code that passes this structure as a parameter to other internal functions. This syntactic change does not affect code quality, but makes the resulting code more readable.

Standard libraries. In comparison to the Java collections library, the Linux kernel and associated C libraries provide limited generic data-structure support. We found that the Java collections library provides a useful set of tools for simplifying driver code. In addition to rewriting the parameter code to use inheritance, we also used Java hash tables in the set-membership tests.

Potential Benefit: Garbage collection. While the E1000 decaf driver currently manages shared objects manually, garbage collection provides a mechanism to simplify this code and prevent resource leaks. When allocating data structures shared between the driver nucleus and decaf driver, the decaf drivers use a custom constructor that also allocates kernel memory at the same time and creates an association in the object tracker.

Rather than relying on the decaf driver to explicitly release this memory, we can write a custom finalizer to free the associated kernel memory when the Java garbage collector frees the object. This approach can simplify exception-handling code and prevent resource leaks on error paths, a common driver problem [31].

Category	Lines of Code Changed
Driver nucleus	381
Decaf driver	4690
User/kernel interface	23

Table 4: Statistics for patches applied to E1000: the lines changed in the driver nucleus, in the decaf driver, and to shared data structures requiring new marshaling code.

5.2 Driver Evolution

We evaluate the ability of Decaf Drivers to support driver evolution by applying all changes made to the E1000 driver between kernel versions 2.6.18.1 and 2.6.27. Because we continue to use the 2.6.18.1 kernel, we omitted the small number of driver changes related to kernel interface updates. We applied all 320 patches in two batches: those before the 2.6.22 kernel and those after. Overall, we found that modifying the driver was simple, and that the driver nucleus and decaf driver could be modified and compiled separately.

The changes are summarized in Table 4. The vast majority of code changes were at user level. Thus, the bulk of the development on E1000 since the 2.6.18.1 kernel would have been performed in Java at user level, rather than in the kernel in C. Furthermore, only 23 changes affected the kernel-user interface, for example by adding or removing fields from shared structures.

In these cases, we modified the kernel implementation of the data structure, and re-split the driver to produce updated versions of the Java data structures. To ensure that new structure fields are marshaled between the driver nucleus and decaf driver, we added one additional annotation for each new field to the original driver. These annotations ensure that DriverSlicer generates marshaling code to allow reading and/or writing the new variables in the decaf driver.

5.3 Mixing C and Java

A substantial portion of the Decaf architecture is devoted to enabling a mix of Java and C to execute at user level. We have found two reasons to support both languages. First, when migrating code to Java, it is convenient to move one function at a time and then test the system, rather than having to convert all functions at once (as required by most user-level driver frameworks). This code is temporary and exists only during the porting process. We initially ran all user-mode E1000 functions in this mode and then incrementally converted them to Java, starting with leaf functions and then advancing up the call graph. Our current implementation has no driver functionality implemented in the driver library.

Jeannie makes this transition phase simple because of its ability to mix Java and C code without explicitly us-

ing the Java Native Interface. The ability to execute either Java or C versions of a function during development greatly simplified conversion, as it allowed us to eliminate any new bugs in our Java implementation by comparing its behavior to that of the original C code.

Second, and more important, there may be functionality necessary for communicating with the kernel or the device that is not possible to express in Java. These functions are *helper routines* that do not contain driver logic but provide an escape from the limits of a managed language. Some examples we have observed include accessing the `sizeof()` operator in C, which is necessary for allocating some kernel data structures, and for performing programmed I/O with I/O ports or memory-mapped I/O regions. While some languages, including C#, support unsafe memory access, Java does not. However, we found that none of these helper routines are specific to the E1000 driver, and as a result placed them in the decaf runtime to be shared with other decaf drivers. As before, Jeannie makes using these helper routines in Java a straightforward matter.

Jeannie also simplifies the user-level stub functions significantly. These stubs include a simple mixture of C and Java code, whereas using JNI directly would significantly complicate the stubs.

6 Related Work

Decaf Drivers differs from past work on driver reliability and type-safe kernel programming in many respects. Unlike past approaches that are either compatible or transformative, we desire both compatibility with existing code and the opportunity to completely rewrite drivers.

Driver reliability. Driver reliability systems focus on tolerating faults in existing drivers with hardware memory protection [45, 50], language-based isolation [52], or private virtual machines [17, 18, 27]. However, these systems all leave driver code in C and in the kernel and thus do not ease driver programming.

Driver safety. Another approach to improving reliability is to prevent drivers from executing unsafe actions. Safety can be achieved by executing drivers in user mode [21, 26], with type safety in the kernel [9], or by formally verifying driver safety [41]. However, these approaches either require writing a completely new driver, or rewriting the entire kernel. With Decaf Drivers, drivers may be incrementally converted to any language because C is still available for helper routines.

Simplifying driver code. Many projects promise to simplify driver programming through new driver interfaces [3, 33, 38, 25, 51, 42, 30]. These systems offer advanced languages [51, 42]; domain-specific languages for hardware access [30] and for common driver logic [7, 10]; simplified programming interfaces at user-

level [3, 8, 33]; and cross-platform interfaces [38, 25]. Like Decaf Drivers, Linux UIO drivers leave part of the driver in the kernel, while the bulk executes at user level [47]. Coccinelle [37] simplifies patching a large set of drivers at once. The features offered by these systems are complementary to Decaf Drivers, and the ability to gradually rewrite driver code in a new language may provide a route to their use. However, these systems either require writing new drivers for a new interface, or they simplify existing drivers but not enough: drivers are left in C and in the kernel.

Type-safe kernel programming. SPIN [23], the J Kernel [48], and Singularity [24] have kernels written in type safe languages. More recently, a real-time JVM was ported to the Solaris kernel [36]. In contrast to these systems, Decaf Drivers enables the use of modern languages for drivers without rewriting or substantially adding to the OS kernel.

7 Conclusion

Device drivers are a major expense and cause of failure for modern operating systems. With Decaf Drivers, we address the root of both problems: writing kernel code in C is hard. The Decaf architecture allows large parts of existing drivers to be rewritten in a better language, and supports incrementally converting existing driver code. Drivers written for Decaf retain the same kernel interface, enabling them to work with unmodified kernels, and can achieve the same performance as kernel drivers. Furthermore, tool support automates much of the task of converting drivers, leaving programmers to address the driver logic but not the logistics of conversion.

Writing drivers in a type-safe language such as Java provides many concrete benefits to driver programmers: improved reliability due to better compiler analysis, simplified programming due to richer runtime libraries, and better error handling with exceptions. In addition, many tools for user-level Java programming may be used for debugging.

Acknowledgments. We would like to thank our shepherd, Robert Grimm, for his useful comments and for his help, along with Martin Hirzel, in resolving Jeannie bugs. This work was supported in part by NSF grant CSR 0745517. Swift has a financial interest in Microsoft Corp.

References

- [1] H. Albrecht. Remote Tea. <http://remotetea.sourceforge.net/>.
- [2] J. Allchin. Windows Vista team blog: Updating a brand-new product, Nov. 2006. <http://windowsvistablog.com/blogs/windowsvista/archive/2006/11/17/updating-a-brand-new-product.aspx>.
- [3] F. Armand. Give a process to your drivers! In *Proc. of the EurOpen Autumn 1991*, Sept. 1991.
- [4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.
- [5] E. Brewer, J. Condit, B. McCloskey, and F. Zhou. Thirty years is long enough: getting beyond c. In *Proc. of the Tenth IEEE HOTOS*, 2005.
- [6] A. Catorcini, B. Grunkemeyer, and B. Grunkemeyer. CLR inside out: Writing reliable .NET code. *MSDN Magazine*, Dec. 2007. <http://msdn2.microsoft.com/en-us/magazine/cc163298.aspx>.
- [7] P. Chandrashekaran, C. Conway, J. M. Joy, and S. K. Rajamani. Programming asynchronous layers with CLARITY. In *Proc. of the 15th ACMFSE*, Sept. 2007.
- [8] P. Chubb. Get more device drivers out of the kernel! In *Ottawa Linux Symp.*, 2004.
- [9] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proc. of the ACM SIGPLAN '03 ACM Conference on Programming Language Design and Implementation*, June 2003.
- [10] C. L. Conway and S. A. Edwards. NDL: a domain-specific language for device drivers. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2004.
- [11] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Associates, Feb. 2005.
- [12] I. N. Division. Netperf: A network performance benchmark. <http://www.netperf.org>.
- [13] M. Eisler. XDR: External data representation standard. RFC 4506, Internet Engineering Task Force, May 2006.
- [14] J. Elson. FUSD: A Linux framework for user-space devices, 2004. User manual for FUSD 1.0.
- [15] D. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. of the 18th ACM SOSP*, Oct. 2001.
- [16] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proc. of the 7th USENIX OSDI*, 2006.
- [17] Ú. Erlingsson, T. Roeder, and T. Wobber. Virtual environments for unreliable extensions. Technical Report MSR-TR-05-82, Microsoft Research, June 2005.
- [18] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.
- [19] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proc. of the Thirteenth ACM ASPLOS*, Mar. 2008.
- [20] B. Goetz. Plugging memory leaks with weak references. <http://www.ibm.com/developerworks/java/library/j-jtp11225/index.html>, 2005.

- [21] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proc. of the 2007 IEEE DSN*, June 2007.
- [22] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Proc. of the ACM OOPSLA '07*, Oct. 2007.
- [23] W. Hsieh, M. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B. Bershad. Language support for extensible operating systems. In *Proc. of the Workshop on Compiler Support for System Software*, Feb. 1996.
- [24] G. Hunt, J. Larus, M. Abadie, M. A. and PP. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. L. and Ni. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.
- [25] Jungo. Windriver cross platform device driver development environment. Technical report, Jungo Corporation, Feb. 2002. <http://www.jungo.com/windriver.html>.
- [26] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 20(5), 2005.
- [27] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. of the 6th USENIX OSDI*, 2004.
- [28] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. of the 6th USENIX OSDI*, 2004.
- [29] R. Love. Kernel locking techniques. *Linux Journal*, Aug. 2002. <http://www.linuxjournal.com/article/5833>.
- [30] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proc. of the 4th USENIX OSDI*, Oct. 2000.
- [31] Microsoft Corp. PREFast for drivers. <http://www.microsoft.com/whdc/devtools/tools/prefast.mspx>.
- [32] Microsoft Corporation. Windows Server 2003 DDK. <http://www.microsoft.com/whdc/DevTools/ddk/default.mspx>, 2003.
- [33] Microsoft Corporation. Architecture of the user-mode driver framework. <http://www.microsoft.com/whdc/driver/wdf/UMDF-arch.mspx>, May 2006. Version 0.7.
- [34] Microsoft Corporation. Interoperating with unmanaged code. [http://msdn.microsoft.com/en-us/library/sd10k43k\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/sd10k43k(VS.71).aspx), 2008.
- [35] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Constr.*, 2002.
- [36] T. Okumura, B. R. Childers, and D. Mosse. Running a Java VM inside an operating system kernel. In *Proc. of the 4th ACM VEE*, Mar. 2008.
- [37] Y. Padoleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proc. of the 2008 EuroSys Conference*, apr 2008.
- [38] Project UDI. Uniform Driver Interface: Introduction to UDI version 1.0. http://udi.certeck.cc/Docs/pdf/UDI_tech-white_paper.pdf, Aug. 1999.
- [39] D. Richie. The Unix tree: the 'nss' kernel, Jan. 1999. <http://minnie.tuhs.org/UnixTree/Nsys/>.
- [40] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Tampering device drivers. In *Proc. of the 2009 EuroSys Conference*, Apr. 2009.
- [41] L. Ryzhyk, I. Kuz, and G. Heiser. Formalising device driver interfaces. In *Proc. of the Workshop on Programming Languages and Systems*, Oct. 2007.
- [42] M. Spear, T. Roeder, O. Hodson, G. Hunt, and S. Levi. Solving the starting problem: Device drivers as self-describing artifacts. In *Proc. of the 2006 EuroSys Conference*, Apr. 2006.
- [43] Sun Microsystems. UNIX programmer's supplementary documents: rpcgen programming guide. <http://docs.freebsd.org/44doc/psd/22.rpcgen/paper.pdf>.
- [44] M. Susskraut and C. Fetzer. Automatically finding and patching bad error handling. In *Proceedings of the Sixth European Dependable Computing Conference*, 2006.
- [45] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), Feb. 2005.
- [46] L. Torvalds. Linux kernel source tree. <http://www.kernel.org>.
- [47] L. Torvalds. UIO: Linux patch for user-mode I/O, July 2007.
- [48] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-Kernel: a capability-based operating system for Java. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*, pages 369–393. Springer-Verlag, 1999.
- [49] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of the 8th USENIX OSDI*, Dec. 2008.
- [50] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proc. of the 20th ACM SOSP*, 2005.
- [51] H. Yamauchi and M. Wolczko. Writing Solaris device drivers in Java. Technical Report TR-2006-156, Sun Microsystems, Apr. 2006.
- [52] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th USENIX OSDI*, 2006.

Rump File Systems: Kernel Code Reborn

Antti Kantee

Helsinki University of Technology

pooka@cs.hut.fi

Abstract

When kernel functionality is desired in userspace, the common approach is to reimplement it for userspace interfaces. We show that use of existing kernel file systems in userspace programs is possible without modifying the kernel file system code base. Two different operating modes are explored: 1) a transparent mode, in which the file system is mounted in the typical fashion by using the kernel code as a userspace server, and 2) a standalone mode, in which applications can use a kernel file system as a library. The first mode provides isolation from the trusted computing base and a secure way for mounting untrusted file systems on a monolithic kernel. The second mode is useful for file system utilities and applications, such as populating an image or viewing the contents without requiring host operating system kernel support. Additional uses for both modes include debugging, development and testing.

The design and implementation of the Runnable Userspace Meta Program file system (*rump fs*) framework for NetBSD is presented. Using rump, ten disk-based file systems, a memory file system, a network file system and a userspace framework file system have been tested to be functional. File system performance for an estimated typical workload is found to be $\pm 5\%$ of kernel performance. The prototype of a similar framework for Linux was also implemented and portability was verified: Linux file systems work on NetBSD and NetBSD file systems work on Linux. Finally, the implementation is shown to be maintainable by examining the 1.5 year period it has been a part of NetBSD.

1 Introduction

Motivation. “*Userspace or kernel?*” A typical case of driver development starts with this exact question. The tradeoffs are classically well-understood: speed, efficiency and stability for the kernel or ease of program-

ming and a more casual development style for userspace. The question stems from the different programming environments offered by the two choices. Even if code written for the kernel is designed to be run in userspace for testing, it is most likely implemented with `#ifdef`, crippled and does not support all features of kernel mode.

Typical operating system kernels offer multitudes of tested and working code with reuse potential. A good illustration is file system code, which in the case of most operating systems also comes with a virtual file system abstraction [18] making access file system independent.

By making kernel file systems function in userspace, existing code can be utilized for free in applications. We accomplished this by creating a shim layer to emulate enough of the kernel to make it possible to link and run the kernel file system code. Additionally, we have created supplementary components necessary to integrate the file system code with a running system, i.e. mount it as a userspace file server. Our scheme requires no modifications to existing kernel file system code.

We define a Runnable Userspace Meta Program file system (*rump fs*) to be kernel file system code used in a userspace application or as a userspace file server.

Results. NetBSD [20] is a free 4.4BSD derived OS running on over 50 platforms and used in the industry especially in embedded systems and servers. A real world usable implementation of rump file systems, included in NetBSD since August 2007, has been written.

The following NetBSD kernel file systems are *usable and mountable in userspace without source modifications*: cd9660, EFS, Ext2fs, FFS, HFS+, LFS, MSDOSFS, NFS (client¹), NTFS, puffs, SysVBFS, tmpfs, and UDF. All are supported from the same code-base without file system specific custom code.

Additionally, a quick prototype of a similar system for the Linux kernel was implemented. Under it, the relatively simple jffs2 [31] journaling file system from the Linux kernel is mountable as a userspace server on NetBSD. Other Linux file systems could also be made

to work using the same scheme, but since they are more complex than jffs2, additional effort would be required.

Finally, we introduce the *fs-utils* suite and an improved *makefs* utility [19]. Both use rump for generic file system access and do not implement private userspace file system drivers. In contrast, software packages such as mtools and e2fsprogs reimplement thousands of lines of file system code to handle a single file system.

Contributions. This paper shows that it is possible and convenient to *run pre-existing kernel file system code in a userspace application*. This approach has been desired before: Yang et al. described it as ideal for their needs but rated implementation hopelessly difficult [32].

We also describe a way to make a monolithic style kernel operate like a multiserver microkernel. In contrast to previous work, our approach *gives the user the choice of micro- or monolithic kernel operation*, thereby avoiding the need for the performance discussion.

The paper also shows it is possible to use kernel code in userspace on top of a POSIX environment irrespective of the kernel platform the code was originally written for. This paves way to thinking about *kernel modules as reusable operating system independent components*.

Userspace file systems. This paper involves file systems in userspace but it is not a paper on userspace fs frameworks. Userspace fs frameworks provide a programming interface for the file server to attach to and a method for transporting file system requests in and out of the kernel. This paper explores running kernel file system code as an application in userspace. Our approach requires a userspace fs framework only in case mounting the resulting rump file system is desired. The choice of the framework is mostly orthogonal. puffs [15] was chosen because of the author's familiarity and because it is the native solution on NetBSD. Similarly, would the focus of implementation have been Linux or Windows NT, the choice could have been FUSE [28] or FIFS [3].

Paper organization. The rest of this paper is organized as follows: Section 2 deals with architecture issues. Some major details of the implementation are discussed in Section 3. The work is measured and evaluated in Section 4. Section 5 surveys related work and finally Section 6 provides conclusions and outlines future work.

2 Architecture

Before going into details about the architecture of the implementation, let us recall how file systems are implemented in a monolithic kernel such as NetBSD or Linux.

- The interface through which the file system is accessed is the virtual file system interface [18]. It provides virtual nodes as abstract objects for accessing files independent of the file system type.

- To access the file system backend, the file system implementation uses the necessary routines from the kernel. These are for example the disk driver for a disk-based file system such as FFS, sockets and the networking stack for NFS or the virtual memory subsystem for tmpfs [27]. Access is usually done through the buffer cache.
- For file content caching and memory mapped I/O a file system is heavily tied to the virtual memory subsystem [25]. In addition to the pager's get and put routines, various supporting routines are required. This integration also provides the page cache.
- Finally, a file system uses various kernel services. Examples range from a hashing algorithm to timer routines and memory allocation. The kernel also performs access brokering and makes sure the same image is not mounted twice.

If the reuse of file system code in userspace is desired, all of these interfaces must be provided in userspace. As most parts of the kernel do not have anything to do with hardware but rather just implement algorithms, they can be simply linked to a userspace program. We define such code to be *environment independent* (EI). On the other hand, for example device drivers, scheduling routines and CPU routines are *environment dependent* (ED) and must be reimplemented.

2.1 Kernel and Userspace Namespace

To be able to understand the general architecture, it is important to note the difference between the namespaces defined by the C headers for kernel and for user code. Selection of the namespace is usually done with the pre-processor, e.g. -D_KERNEL. Any given module must be *compiled* in either the kernel or user namespace. After compilation the modules from different namespaces can be *linked* together, assuming that the application binary interface (ABI) is the same.

To emulate the kernel, we must be able to make user namespace calls, such as memory allocation and I/O. However, code cannot use kernel and user namespaces simultaneously due to collisions. For example, on a BSD-based system, the libc `malloc()` takes one parameter while the kernel interface takes three. To solve the problem, we identify components which require the kernel namespace and components which require the user namespace and compile them as separate compilation units. We let the linker handle unification.

The namespace collision issue is even more severe if we wish to use rump file systems on foreign platforms. We cannot depend on anything in the NetBSD kernel namespace to be available on other systems. Worse,

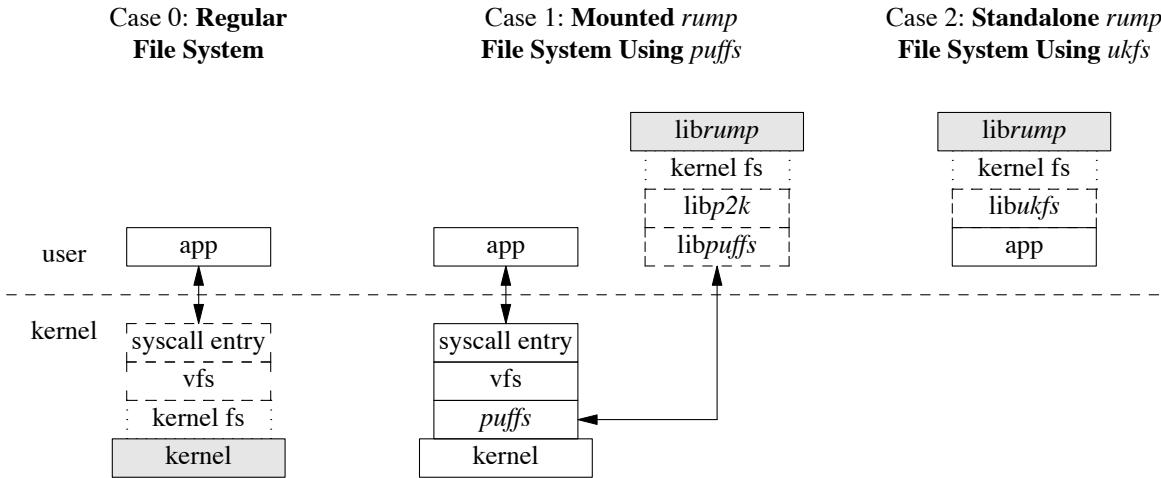


Figure 1: Rump File System Architecture

we cannot include any headers from the NetBSD kernel namespace in applications on other platforms, since it will create conflicts. For example, think what will happen if an application includes both the native and NetBSD `<sys/stat.h>`. To address this, we provide a namespace which applications can use to make calls to the rump kernel namespace. For example, the kernel vnode operation `VOP_READ()` is available under the name `RUMP_VOP_READ()`.

2.2 Component Overview

The architecture of the framework is presented in Figure 1 using three different cases to illuminate the situation. Analogous parts between the three are signaled. The differences are briefly discussed below before moving to further dissect the components and architecture.

Regular File System (Case 0). To give context, “Regular File System” shows a file system in the kernel. Requests from an application are routed through the system call layer and vfs to be handled by the file system driver.

Mounted rump File System Using puffs (Case 1). From the application perspective, a mounted rump file system behaves like the same file system running in the kernel. The NetBSD userspace file systems framework, puffs [15], is used to attach the file system to the kernel.

Standalone rump File System Using ukfs (Case 2). A standalone rump file system is not mounted into the file system namespace. Rather, applications use a special API to mount and access the file system. While this requires code level awareness, it allows complete freedom, including the ability to target a certain file system and make calls past the virtual file system abstraction. The key benefits of standalone rump file systems are that they do not require kernel support or the use of operations normally reserved for the superuser, e.g. `mount()`.

```
int rumpuser_gettimeofday(struct timeval *tv,
                         int *error);

ssize_t rumpuser_pread(int fd, void *buf,
                      size_t bufsize, off_t offset, int *error);

int rumpuser_thread_create(void *(*f)(void*), void*);

void rumpuser_mutex_enter(struct rumpuser_mtx *);
```

Figure 2: Examples of *rumpuser* interfaces

2.3 The File System Driver

The file system driver is compiled as a regular userspace shared library. It can be linked directly into the file server or loaded dynamically at runtime using `dlopen()`.

2.4 librump

The interfaces required by a file system were classified in the beginning of Section 2. The component to provide these interfaces in the absence of the real kernel is librump. It emulates enough of the kernel environment for the file system code to be able to run.

To solve the namespace problem described in Section 2.1, librump is split into two: rumpkern and rumpuser. The first is compiled as a kernel component and the latter as a userspace component. Both must be linked into rump file systems.

Figure 2 presents examples of routines provided by rumpuser. There are two main classes of calls provided by rumpuser: system calls and thread library calls. Additionally, support calls such as memory allocation exist.

A convenient observation is to note that the file systems only call routines within themselves and interfaces in our case provided by rumpkern. Rumpkern only calls routines within itself, the file system (via callbacks) and

Component	# of lines
rumpuser	491
rumpkern (ED)	3552
std kern (EI)	27137
puffs (kernel)	3411
FFS	14912

Table 1: rump library size analysis

rumpuser. Therefore, by closure, rumpuser is the component defining the portability of a rump file system.

Librump was engineered bottom-up to provide kernel interfaces for file systems. Most of the interfaces could be used from the kernel source tree as such (environment independent), but some had to be reimplemented for userspace (environment dependent). For example, the vm subsystem was completely reimplemented for rump and could be simplified from tens of thousands of lines of code to just hundreds of lines because of most vm functionality being irrelevant in userspace. Table 1 shows effort in lines of code without comments or empty lines. Two kernel file systems are included for comparison. Code size is revisited in Section 4.5.

2.5 libp2k

Mounted rump file systems (Case 1, Figure 1) use the NetBSD userspace file systems framework, puffs [15]. We rely on two key features. The first one is transporting file system requests to userspace, calling the file server, and sending the results back. The second one is handling an abruptly unmounted file system, such as a file server crash. The latter prevents any kernel damage in the case of a misbehaving file server.

puffs provides an interface for implementing a userspace file systems. While this interface is clearly heavily influenced by the virtual file system interface, there are multiple differences. For example, the kernel virtual file system identifies files by a struct vnode pointer, whereas puffs identifies files using a puffs_cookie_t value. Another example of a parameter difference is the struct uio parameter. In the kernel this is used to inform the file system how much data to copy and in which address space. puffs passes this information to the read interface as a pointer where to copy to along with the byte count - the address space would make no difference since a normal userspace process can only copy to addresses mapped in its vmspace. In both cases the main idea is the same but details differ.

The *p2k*, or puffs-to-kernel, library is a request translator between the puffs userspace file system interface and the kernel virtual file system interface. It also interprets the results from the kernel file systems and converts them

```

int
p2k_node_read(struct puffs_usermount *pu,
    puffs_cookie_t opc, uint8_t *buf,
    off_t offset, size_t *resid,
    const struct puffs_cred *pcr, int ioflag)
{
    kauth_cred_t cred;
    struct uio *uiو;
    int rv;

    cred = cred_create(pcr);
    uiو = rump_uio_setup(buf, *resid, offset,
        RUMPUIO_READ);
    VLS(opc);
    rv = RUMP_VOP_READ(opc, uiو, ioflag, cred);
    VUL(opc);
    *resid = rump_uio_free(uiو);
    cred_destroy(cred);
    return rv;
}

```

Figure 3: p2k_node_read() Implementation

back to a format that puffs understands.

To give an example of p2k operation, we discuss reading a file. This is illustrated by the p2k read routine in Figure 3. We see the uiو structure created by rump_uio_setup() before calling the vnode operation and freed after the call while saving the results. We also notice the puffs credit type being converted to the opaque kauth_cred_t type used in the kernel. This is done by the p2k library's cred_create() routine, which in turn uses rump_cred_create(). The VLS() and VUL() macros in p2k to deal with NetBSD kernel virtual file system locking protocol. They take a shared (read) lock on the vnode and unlock it, respectively.

Mount utilities and file servers

Standard kernel file systems are mounted with utilities such as mount_efs, mount_tmpfs, etc. These utilities parse the command line arguments and call the mount() system call.

Our equivalent mountable rump file system counterparts are called rump_efs, rump_tmpfs, etc. Instead of calling the regular mount call, they attach to the system by p2k and rump. To maximize integration, these file servers share the same command line argument parsing code with the regular mount utilities. This was done by restructuring the mount utilities to provide an interface for command line argument parsing.

Sharing the argument parsing means that the file servers have the same syntax and makes usage interchangeable just by altering the command name. We also modified the system to handle a rump option in /etc/fstab. This allows to toggle certain mount-points such as USB devices and CD/DVD to be handled using rump file systems by just adding one option.

```

struct ukfs *ukfs_mount(const char *fstype,
    const char *devpath, const char *mntpath,
    int mntflag, void *arg, size_t arglen);

int ukfs_modload(const char *libpath);
int ukfs_modload_dir(const char *directory);

ssize_t ukfs_read(struct ukfs *u, const char *file,
    off_t off, uint8_t *buf, size_t bufsize);

int ukfs_rmdir(struct ukfs *u, const char *dir);

```

Figure 4: Examples of ukfs interfaces

2.6 libukfs

The ukfs library, or user-kernel file system, provides a standalone approach (Case 2 from Figure 1). Two classes of interfaces are provided by libukfs, both having examples in Figure 4, and are discussed below:

Initialization. To use a file system, it must be virtually mounted. The mount call returns a `struct ukfs` handle which is passed to all other calls. This handle is analogous to the mountpoint path in a mounted file system.

Additionally, routines for dynamically loading file system libraries are provided. This is similar to loading kernel modules, but since we are in userspace, `dlopen()` is used for loading.

File system access. Accessing file system contents is done with calls in this class. Most calls have an interface similar to system calls, but as they are self-contained, they take a filename instead of for example requiring a separate open before passing a file descriptor to a call. The rootpath is the root of the file system, but the library provides tracking of the current working directory, so passing non-absolute paths is possible.

If an application wishes to do low level calls such as vfs operations for performance or functionality reasons, it is free to do so even if it additionally uses ukfs routines.

3 Implementation

This section deals with major points of interest in the implementation. While the discussion is written with NetBSD terminology, it attempts to take into account the general case with all operating systems.

3.1 Disk Driver

A disk block device driver provides storage medium access and is instrumental to the operation of disk-based file systems. The main interface is simple: a request instructs the driver to read or write a given number of sectors at a given offset. The disk driver queues the request and returns. The request is handled in an order according to a set policy, e.g. the disk head elevator. The request

must be handled in a timely manner, since during the period that the disk driver is handling the request the object the data belongs to (e.g. vm page) is held locked. Once the request is complete, the driver signals the kernel that the request has been completed. In case the caller waits for the request to complete, the request is said to be synchronous, otherwise asynchronous.

There are two types of backends: buffered and unbuffered. A buffered backend stores writes to a buffer and flushes them to storage later. An unbuffered backend will write to storage immediately. Examples are a regular file and a character device representing a disk partition, respectively. A block driver signals a completed write only after data has hit the disk. This means that a disk driver operating in userspace must make sure the data is not still in a kernel cache before it issues the signal.

There are three approaches to implementing the block driver using standard userspace interfaces.

- **Use `read()` and `write()` in caller context:** this is the simplest method. However, it effectively makes all requests synchronous and kills write performance.
- **Asynchronous read/write:** in this model the request is handed off to an I/O thread. When the request has been completed, the I/O thread issues an “interrupt” to signal completion.

A buffered backend must flush synchronously executed writes. The only standard interface available for this is `fsync()`. However, it will flush all buffered data before returning, including previous asynchronous writes. Non-standard ranged interfaces such as `fsync_range()` exist, but they usually flush at least some file metadata in addition to the actual data causing extra unnecessary I/O.

A userlevel write to an unbuffered backend goes directly to storage. The system call will return only after the write has been completed. No flushing is required, but since userlevel I/O is serialized in Unix, it is not possible to issue another write before the first one finishes. This means that a synchronous write must block and wait until an ongoing asynchronous write has been fully executed.

The `O_DIRECT` file descriptor flag causes a write on a buffered backend to bypass cache and go directly to storage. The use of the flag also invalidates the cache for the written range, so it is safe to use in conjunction with buffered I/O. However, the flag is advisory. If conditions are not met, the I/O will silently fall back to the buffer. This method can therefore be used only when it applies for sure.

- **Memory-mapped I/O:** this method works only for regular files. The benefits are that the medium ac-

cess fastpath does not involve any system calls and that the `msync()` system call can be portably used to flush ranges instead of the whole memory cache.

The file can be mapped using windows. This provides two advantages. First, files larger than the available VA can be used. Second, in case of a crash, the core dump is only increased by the size of the windows instead of the size of the image. This is a very important pragmatic benefit. We found that the number of windows does not make a huge difference; we default to 16 1MB windows with LRU.

The downside of the memory mapping approach is that to overwrite data, the contents must first be paged in, then modified, and only after that written. This is to be contrasted to explicit I/O requests, where it is possible to decide if a whole page is being overwritten, and skip pagein before overwrite.

Of the above, we found that on buffered backends `O_DIRECT` works best. Ranged syncing and memory mapped I/O have roughly equal performance and full syncing performs poorly. The disk driver question is revisited in Section 4.6, where we compare performance against a kernel mount.

3.2 Locking and Multithreading

File systems make use of locking to avoid data corruption. Most file systems do not create separate threads, but use the context of the requesting thread to do the operations. In case of multiple requests there may be multiple threads in the file system and they must synchronize access. Also, some file systems create explicit threads, e.g. for garbage collection.

To support multithreaded file systems in userspace, we must solve various subproblems: locks, threads, legacy interfaces and the kernel giantlock. We rely on the userspace pthread library instead of implementing our own set of multithreading and synchronization routines.

Locks and condition variables. There are three different primitives in NetBSD: mutexes, rwlocks and condition variables. These map to pthread interfaces. The only differences are that the kernel routines are of type `void` while the pthread routines return a success value. However, an error from e.g. `pthread_mutex_lock()` means a program bug such as deadlock and in case of failure, the program is aborted and core is dumped.

Threads. The kernel provides interfaces to create and destroy threads. Apart from esoteric arguments such as binding the thread to a specific CPU, which we ignore, the kernel interfaces can be mapped to a pthread library calls. This means that `kthread_create()` will call `pthread_create()` with suitable arguments.

Kernel giantlock. Parts of the NetBSD kernel not converted to fine grained locking are still under the kernel biglock. This lock is special, as it is recursive and must be completely released when blocking. As all the system calls rump makes are in rumpuser, the blocking points are there also. We wrap the potentially blocking calls to drop and retake the biglock.

Legacy interfaces A historic BSD interface still in use in some parts of the kernel is `tsleep()`. It is a facility for waiting for events and maps to pthread condition variables.

Observations. It is clear that the NetBSD kernel and pthread locking and threading interfaces are very close to each other. However, there are minimal differences such as the naming and of course under the hood the implementations are different. Providing a common interface for both [8] would be a worthwhile exercise in engineering for a platform where this was not considered initially.

3.3 puffs as a rump file system

Using rump, puffs can be run in userspace on top of rump and a regular userspace file system on top of it. It gives the benefit of being able to access any file system via ukfs, regardless of whether it is a kernel file system or a userspace file system. Since puffs provides emulation for the FUSE interface [16] as well, any FUSE file system is usable through the same interface too. For instance, a utility which lists the directory tree of a file system works regardless of if the file system is the NetBSD kernel FFS or FUSE ntfs-3g.

Naturally, it would be possible to call userspace file system interfaces from applications without a system as complex as rump. However, since we already do have rump, we can provide total integration for all file systems with this scheme. It would be entirely possible to make ukfs use different callpaths based on the type of file system used. However, that would require protocol conversion in ukfs to e.g. FUSE. Since the puffs stack already speaks all the necessary protocols, it is more elegant to run everything through it.

3.4 Installation and Linking

Rump file systems are installed for userspace consumers as a number of separate libraries. The base libraries are: librump (rumpkern), librumpuser (rumpuser), libukfs and libp2k. Additionally, there are all the individual file system drivers such as librumpfs_efs, librumpfs_ntfs and so forth. To use rump file systems, the base libraries should be linked in during compilation. The file system driver libraries may be linked in during compilation or loaded dynamically.

The NetBSD kernel expects all built-in file systems to be stored in a *link set* for bootstrap initialization. A link set is a method for a source module to store information to a specific section in the object. A static linker unifies the section contents from all source modules into a link set when the program is linked. However, this scheme is not fully compatible with dynamic linking: the dynamic loader would have to create storage to accommodate for section information from each shared library. We discovered that a link set entry only from the first shared library on the linker command line is present runtime. We could have attempted to modify the dynamic linker to support this non-standard feature, but instead we chose to require dynamic loading of file systems when support for more than one is required. Loading is done using the ukfs interfaces described in Section 2.6.

Since the kernel environment is in constant flux, the standard choice of bumping the major library version for each ABI change did not seem reasonable. Instead, currently the compatibility between librump and the file system libraries is handled exactly like for kernel modules: both librump and the file system libraries are embedded with the ABI version they were built against. When a file system library is attached to librump the versions are compared and if incompatible the attach routine returns EPROGMISMATCH.

3.5 Foreign Platforms

Different kernel version. An implication of rump file systems is the ability to use file system code from a different OS version. While it is possible to load and unload kernel modules on the fly, they are closely tied by the kernel ABI. Since a rump file system is a self-contained userspace entity, it is possible to use a file system from a newer or older kernel. Reasons include taking advantage of a new file system feature without having to reboot or avoiding a bug present in newer code.

NetBSD rump file systems on Linux. This means using NetBSD kernel file systems on a Linux platform. As Linux does not support puffs, libp2k cannot be used. A port to FUSE would be required. Despite this, the file system code can be used via ukfs and accessing a file system using NetBSD kernel code on Linux has been verified to work. A notable fact is that structures are returned from ukfs using the ABI from the file system platform, e.g. `struct dirent` is in NetBSD format and must be interpreted by callers as such. Eventually, a component translating structures between different operating systems will be provided.

Linux kernel file systems on NetBSD. Running Linux kernel file systems on NetBSD is interesting because there are several file systems written against the Linux kernel which are not available natively in NetBSD or in

more portable environments such as userspace via FUSE. Currently, our prototype Linux implementation supports only jffs2 [31]. This file system was chosen as the initial target because of its relative simplicity and because it has potential real-world use in NetBSD, as NetBSD lacks a wear-leveling flash file system.

An emulation library targeted for Linux kernel interfaces, *lump*, was created from scratch. In addition, a driver emulating the MTD flash interface used by jffs2 for the backend storage was implemented.

Finally, analogous to libp2k, we had to match the interface of puffs to the Linux kernel virtual file system interface. The main difference was that the Linux kernel has the *dcache* name cache layer in front of the virtual file system nodes instead of being controlled from within each file system individually. Other tasks were straightforward, such as converting the `struct kstat` type received from Linux to the `struct vattr` type expected by puffs and the NetBSD kernel.

ABI Considerations. Linking objects compiled against NetBSD headers to code compiled with Linux headers is strictly speaking not correct: there are no guarantees that the application binary interfaces for both are identical and will therefore work when linked together. However, the only problem encountered when testing on i386 hardware was related to the `off_t` type. On Linux, `off_t` is 32bit by default, while it is 64bit on NetBSD. Making the type 64bit on Linux made everything work.

If mixing components from different NetBSD versions, care must be taken. For example, `time_t` in NetBSD was recently changed from 32bit to 64bit. We must translate `time_t` in calls crossing this boundary.

4 Evaluation

To evaluate the usefulness of rump file systems, we discuss them from the perspectives of security, development uses, application uses, maintenance cost, and performance. We estimate the differences between a rump environment and a real kernel environment and the impact of the differences and provide anecdotal information on fixing several real world bugs using rump file systems.

4.1 Security

General purpose OS file systems are commonly written assuming that file system images contain trusted input. While this was true long ago, in the age of USB sticks and DVDs it no longer holds. Still, users mount untrusted file systems using kernel code. The BSD and Linux manual pages for `mount` warn: “*It is possible for a corrupted file system to cause a crash*”. Worse, arbitrary memory access is known to be possible and fixing each file system to be bullet-proof is at best extremely hard [32].

In a mounted rump file system the code dealing with the untrusted image is isolated in its own process, thus mitigating an attack. As was seen in Table 1, the size difference between a real kernel file system and the kernel portion of puffs is considerable, about five-fold. Since an OS usually supports more than one file system, the real code size difference is much higher. Additionally, puffs was written from ground up to deal with untrusted input.

To give an example of a useful scenario, a recent mailing list posting described a problem with mounting a FAT file system from a USB stick causing a kernel crash. By using a mountable rump file system, this problem was reduced to an application core dump. The problematic image was received from the reporter and problem in the kernel file system code was debugged and dealt with.

```
golem> rump_msdos ~/img/msdosfs.img /mnt
panic: buf mem pool index 23
Abort (core dumped)
golem>
```

4.2 Development and Debugging

Anyone who has ever done kernel development knows that the kernel is not the most pleasant environment for debugging and iteration. A common approach is to first develop the algorithms in userspace and later integrate them into the kernel, but this adds an extra phase.

The following items capture ways in which rump file systems are superior to any single existing method.

- **No separate development cycle:** There is no need to prototype with an ad-hoc userspace implementation before writing kernel code.
- **Same environment:** userspace operating systems and emulators provide a separate environment. Migrating applications (e.g. OpenOffice or FireFox) and network connections there may be challenging. Since rump integrates as a mountable file system on the development host, this problem does not exist.
- **No bit-rot:** There is no maintenance cost for case-specific userspace code because it does not exist.
- **Short test cycle:** The code-recompile-test cycle time is short and a crash results in a core dump and inaccessible files, not a kernel panic and total application failures.
- **Userspace tools:** dynamic analysis tools such as Valgrind [21] can be used to instrument the code. A normal debugger can be used.
- **Complete isolation:** Changing interface behavior for e.g. fault and crash injection [14, 23] purposes can be done without worrying about bringing the whole system down.

To give an example, support for allocating an in-fs journal was added to NetBSD ffs journaling. The author, Simon Burge, is a kernel developer who normally does not work on file systems. He used rump and ukfs for development and described the process thusly: “Instead of rebooting with a new kernel to test new code, I was just able to run a simple program, and debug any issues with gdb. It was also a lot safer working on a simple file system image in a file.” [4].

Another benefit is prototyping. One of the reasons for implementing the 4.4BSD log-structured file system cleaner in userspace was the ability to easily try different cleaning algorithms [24]. Using rump file systems this can easily be done without having to split the runtime environment and pay the overhead for easy development during production use.

Although it is impossible to measure the ease of development by any formal method, we would like to draw the following analogy: kernel development on real hardware is to using emulators as using emulators is to developing as a userspace program.

Differences between environments

rump file systems do not duplicate all corner cases accurately with respect to the kernel. For example, Zhang and Ghose [34] list problems related to flushing resources as the challenging implementation issues with using BSD VFS. Theoretically, flushing behavior can be different if the file system code is running in userspace, and therefore bugs might be left unnoticed. On the flip-side, the potentially different behavior exposes bugs otherwise very hard to detect when running in the kernel. Rump file systems do not possess exactly the same timing properties and details of the real kernel environment. Our position is that this is not an issue.

Differences can also be a benefit. Varying usage patterns can expose bugs where they were hidden before. For example, the recent NetBSD problem report² kern/38057 described a FFS bug which occurs when the file system device node is not on FFS itself, e.g. /dev on tmpfs. Commonly, /dev is on FFS, so regular use did not trigger the problem. However, since this does not hold when using FFS through rump, the problem was triggered more easily. In fact, this problem was discovered by the author while working on the aforementioned journaling support by using rump file systems.

Another bug which triggered much more frequently by using rump file systems was a race which involved taking a socket lock in the nfs timer and the data being modified while blocking for the socket lock. This bug was originally described by the author in a kernel mailing list post entitled “how can the nfs timer work?”. It caused the author to be pointed at a longstanding NFS

problem of unknown cause described in kern/38669. A more detailed report was later filed under kern/40491 and the problem subsequently fixed.

In our final example the kernel FAT file system driver used to ignore an out-of-space error when extending a file. The effect was that written data was accepted into the page cache, but could not be paged out to disk and was discarded without flagging an application error. The rump vnode pager is much less forgiving than the kernel vnode pager and panics if it does not find blocks which it can legally assume to be present. This drew attention to the problem and it was fixed by the author in revision 1.53 of the source module msdosfs_vnops.c.

Locks: Bohrbugs and Heisenbugs

Next we describe cases in which rump file systems have been used to debug real world file system locking problems in NetBSD.

The most reliably repeatable bugs in a kernel environment and a rump file system are ones which depend only on the input parameters and are independent of the environment and timing. Problem report kern/38219 described a situation where the tmpfs memory file system would try to lock against itself if given suitable arguments. This made it possible for an unprivileged user to panic the kernel with a simple program. A problem described in kern/41006 caused a dangling lock when the mknod() system call was called with certain parameters. Both cases were reproduced by running a regular test program against a mounted rump file systems, debugged, fixed and tested.

Triggering race conditions depends on being able to repeat timing details. Problem report kern/40948 described a bug which turned out to be a locking problem in an error branch of the FFS rename vnode operation. It was triggered when the rename source file was removed halfway through the operation. While this is a race condition, it was equally triggerable by using a kernel file system and a mounted rump file system. After being debugged by using rump and fixed, the same problem was reported for tmpfs in kern/41128. It was similarly debugged and dealt with.

Even if the situation depends on components not available in rump file systems, using rump may be helpful. Problem report kern/40389 described a bug which caused a race condition deadlock between the file system driver and the virtual memory code due to lock order reversal. The author wrote a patch which addressed the issue in the file system driver, but did not have a system for full testing available at that time. The suggested patch was tested by simulating the condition in rump. Later, when it was tested by another person in a real environment, the patch worked as expected.

Preventing undead bugs with regression testing

When a bug is fixed, it is good practice to make sure it does not resurface [17] by writing a regression test.

In the case of kernel regression tests, the test is commonly run against a live kernel. This means that to run the test, the test setup must first be upgraded with the test kernel, bootstrapped, and only then can the test be executed. In case the test kernel crashes, it is difficult to get an automatic report in batch testing.

Using a virtual machine helps a little, but issues still remain. Consider a casual open source developer who adds a feature or fixes a bug and to run the regression tests must 1) download or create a full OS configuration 2) upgrade the installation kernel and test programs 3) run tests. Most likely steps “1” and “2” will involve manual work and lead to a diminished likelihood of testing.

Standalone rump file systems are standalone programs, so they do not have the above mentioned setup complications. In addition to the test program, file system tests require an image to mount. This can be solved by creating a file system image dynamically in the test program and removing it once the test is done.

For example, our regression test for the ffs rename race (kern/40948) creates a 5MB FFS image to a regular file and runs the test against it. If the test survives for 10 seconds without crashing, it is deemed as successful. Otherwise, an error is reported:

```
Tests root: /srcs/tests/fs/ffs
t_renamerace (1/1): 1 test cases
    renamerace: Failed: Test case did not exit
                           cleanly: Abort trap (core dumped)
```

4.3 File system access utilities: fs-utils

Several application suites exist for accessing and modifying file system images purely from userspace programs without having to mount the image. For example, Mtools accesses FAT file systems, ntfsprogs is used with NTFS and LTOOLS can access Ext2/3 and ReiserFS. While these suites generally provide the functionality of POSIX command line file utilities such as ls and cp, the name and usage of each command varies from suite to suite.

The fs-utils [33] suite envisioned by the author and implemented by Arnaud Ysmal has been done using standalone rump file systems. The utilities provide file system independent command line utilities with the same functionality and usage as the POSIX counterparts. The code from the POSIX utilities were used as a base for the implementation and the I/O calls were modified from system calls to ukfs calls.

For example, the fsu_ls program acts like regular ls and provides the same 31 flags as ls. The command fsu_ls ~/img/ffs2.img -laF temp produces

the long listing of the contents of the “/temp” directory of an FFS image located in the user’s home directory and `fsu_ls /dev/rsd0e -laF temp` does the same for a FAT located on a USB stick. The file system type is autodetected based on the image contents. Other examples of utilities provided by fs-utils are `fsu_cat`, `fsu_find`, `fsu_chown` and `fsu_diff`.

Additionally, fs-utils provides utilities which are necessary to move data over the barrier between the host system fs namespace and the image. To illustrate the problem, let us consider `fsu_cp`. Like with `cp`, all pathnames given to it are either relative to the current working directory or absolute with respect to the root directory. Since for a standalone rump file system the root directory is the file system image root directory, `fsu_cp` can be used only to copy files within the image. Conversely, command output redirection (`>`) will write to a file on the host, not the image. To remedy these problems, fs-utils provides utilities such as `fsu_ecp`, which copies files over the boundary, as well as `fsu_write`, which reads from standard input and writes to a given file in the file system. For example, the command `ls | fsu_write ~/img/ffs2.img ls.txt` “redirects” the output of `ls` to the file `/ls.txt` on the image.

4.4 makefs

NetBSD is fully cross-compilable without superuser privileges on a POSIX system [19]. This capability is commonly referred to as *build.sh* after the shell script which bootstraps the build process. For the system to be cross-buildable the build process cannot rely on any non-standard kernel functionality to be available, since it might not exist on a non-NetBSD build host.

The canonical way to build a file system image for boot media used to be to create a regular file, mount it using the loopback driver, copy the files to the file system and unmount the image. This required the target file system to be supported on the build host and was not compatible with the goals of *build.sh*.

When *build.sh* was originally introduced to NetBSD, it came with a tool called *makefs*, which creates a file system image from a given directory tree using only application code. In other words, the *makefs* application contains the file system driver. This approach does not require privileges to mount a file system or support for the target file system in the kernel. The original utility had support for Berkeley FFS and was implemented by modifying and reimplementing the FFS kernel code to be able to run in userspace. This was the only good approach available at the time. Support for the ISO9660 CD file system was added later.

The process of *makefs* consists of four phases:

1. scan the source directory

	original	rump
FFS SLOC	1748	247
supported file systems	FFS, iso9660	FFS, ext2, iso9660, FAT, SysVBFS
FFS effort	> 2.5 weeks or 100 hours	2 hours
total effort	7 weeks or 280 hours	2 days or 16 hours

Table 2: Comparison between original and rump makefs. Implementation effort for the original was provided by Luke Mewburn, the author of the original utility. The FFS figure stands only for driver implementation and does not include additional time spent debugging.

2. calculate target image size based on scan data
3. create the target image
4. copy source directory files to the target image

In the original version of *makefs* all of the phases as implemented in a single C program. Notably, phase 4 is the only one that requires a duplicate implementation of features offered by the kernel file system driver.

For comparison, we have implemented *makefs* using kernel file system drivers for phase 4. It is currently available as an unofficial alternative to the original *makefs*. We partially reuse code from the original *makefs*, since we need to analyze the source tree to determine the image size (phases 1&2). We rely on an external *newfs/mkfs* program for creating an empty file system image (phase 3). For phase 4 we use fs-utils and the `fsu_put` utility, which copies a directory hierarchy to a file system image. The exception is ISO9660 support, for which we use the original *makefs* utility; the kernel CD file system driver is read-only.

For phase 3, we had to make sure that the *mkfs/newfs* utility can create a file system to a regular file – typically such utilities operate on device special files. Out of the supported file systems, we had to add support for this to the NetBSD FAT and SysVBFS utilities. Support for each was approximately 100 lines of modification.

We compare the two implementations in Table 2. As can be observed, over a third of the original effort was for implementing support for a single file system driver. Since we reuse the kernel driver, we get this functionality for free. Additionally, `fsu_put` from fs-utils could be used as such. All of the FFS code for the rump implementation is involved in calculating the image size and was available from *makefs*. If code for this had not been available, we most likely would have implemented it using shell utilities. However, since determining the size involves multiple calculations such as dealing with hard links and rounding up directory entry sizes, we concluded that reusing working code was a better option.

Total commits to the kernel	9640
Total commits to rump	438
Commits touching only rump	347
Build fixes	17
Functionality fixes	5
Unique committers	30

Table 3: Commit analysis for the rump source tree from August 2007 to December 2008.

4.5 Maintaining rump in NetBSD

As rump implements environment dependent code in parallel with the the kernel, the implementation needs to keep up. There are two kinds breakage: the kind resulting in compile failure and the kind resulting in non-functional compiled code. The numbers in Table 3 have been collected from version control logs from the period August 2007 - December 2008, during which rump has been part of the official NetBSD source tree. The commits represent the number of changes on the main trunk.

The number of build fixes is calculated from the amount of commits that were done after the kernel was changed and rump not build anymore as a result. For example, a file system being changed to require a kernel interface not yet supported by rump is this kind of failure. Commits in which rump was patched along with the kernel proper were not counted in with this figure.

Similarly, functionality fixes include changes to kernel interfaces which prevented rump from working, in other words the build worked but running the code failed. Regular bugs are not included in this figure.

Unique committers represents the number of people from the NetBSD community who committed changes to the rump tree. The most common case was to keep up with changes in other parts of the kernel.

Based on observations, the most important factor in keeping rump functional in a changing kernel is educating developers about its existence and how to test it. Initially there was a lot of confusion in the community about how to test rump, but things have since gotten better.

It should be kept in mind that over the same time frame the NetBSD kernel underwent very heavy restructuring to better support multicore. As it was the heaviest set of changes over the past 15 years, the data should be considered “worst case” instead of “typical case”.

For an idea of how much code there is to maintain, Figure 5 displays the number of lines of code lines in for rump in the NetBSD source tree. The count is without empty lines or comments. The number of lines of environment dependent code (rumpkern + rumpuser) has gone up from 1443 to 4043 (281%) while the number of code lines used directly from the kernel has gone up from 2894 to 27137 (938%). Features have been added, but

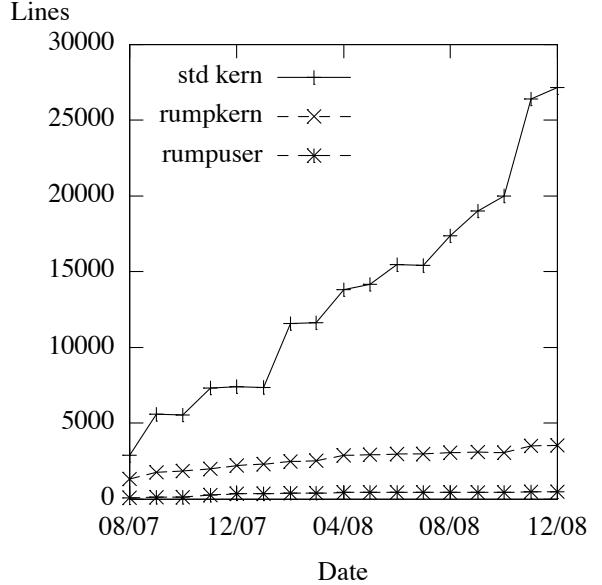


Figure 5: Lines of Code History

much of this has been done with environment independent code. Not only does this reduce code duplication, but it makes rump file systems behave closer to kernel file systems on a detailed level.

There have been two steep increases in code size. The first one was in January 2008, when all of the custom file system code written for userspace, such as namei, was replaced with kernel code. While functionality provided by the interfaces remained equivalent, the special case implementation for userspace was much smaller than the more general kernel code. The general code also required more complex emulation. The second big increase was in October 2008, when the kernel TCP/IP networking stack and support for socket I/O was added to rumpkern.

4.6 Performance

We measure the performance of three macro level operations: directory traversal with `ls -lR`, recursively copying a directory hierarchy containing both small and large files with `cp -R` and copying a large file with `cp`. For testing rump, standalone rump file systems were used with `fs-utils`. Mounted rump file systems were not measured, as they mostly test the performance of puffs and its kernel cache. For the copy operations the source data was precached. The figures are the duration from mount to operation to umount.

The hardware used was a 2GHz Core2Duo PC laptop with a 100GB ATA drive. We performed the measurements on a 4GB FFS disk image hosted on a regular file and a 20GB FFS partition directly on the hard disk. Both file systems were aged [26]: the first one artificially

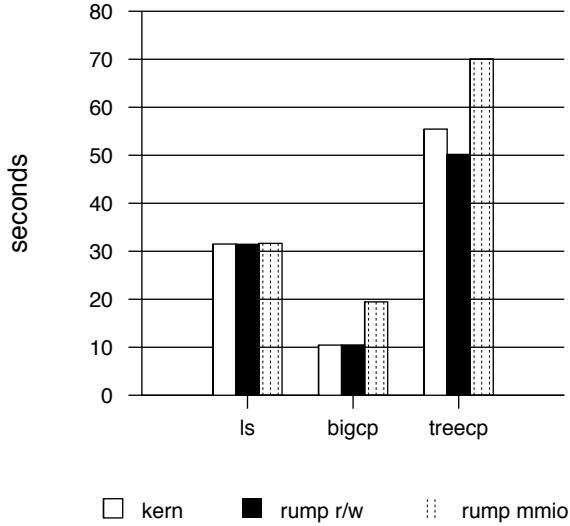


Figure 6: FFS on a regular file (buffered). rump r/w uses direct I/O, as discussed in Section 3.1.

by copying and deleting files. The latter one is in daily use on the author’s laptop and has aged through natural use. The file systems were always mounted so that I/O is performed in the classic manner, i.e. FFS integrity is maintained by performing key metadata operations synchronously. This is to exacerbate the issues with a mix of async and sync I/O requests.

The results are presents in Figure 6 and Figure 7. The figures between the graphs are not directly comparable, as the file systems have a different layout and different aging. The CD image used for the large copy and the kernel source tree used for the treecopy are the same. The file systems have different contents, so the listing figures are not comparable at all.

Analysis. The results are in line with the expectations.

- The directory traversal shows that the read operations perform roughly the same on a regular file and 6% slower for an unbuffered backend. This difference is explained by the fact that the buffered file includes read ahead for a userspace consumer, while the kernel mount accesses the disk unbuffered.
- Copying the large file is 98.5% asynchronous data writes. Memory mapped I/O is almost twice as slow as read/write, since as explained in Section 3.1, the relevant parts of the image must be paged in before they can be overwritten and thus I/O bandwidth requirement is double. Unbuffered userspace read/write is 1.5% slower than the kernel mount.
- Copying a directory tree is a mix of directory metadata and file data operations and one third of the

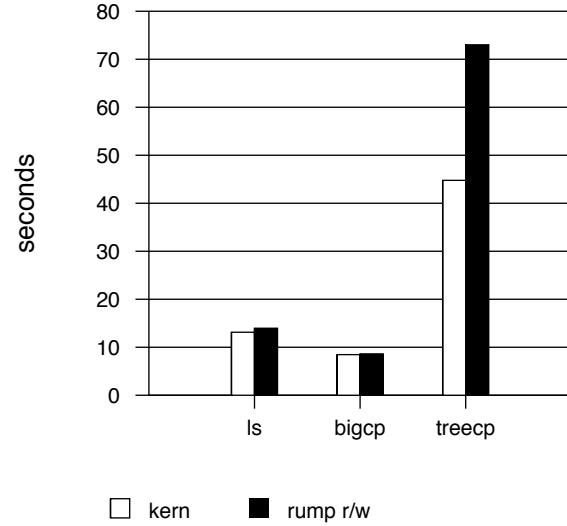


Figure 7: FFS on a HD partition (unbuffered), accessed through a character device.

I/O is done synchronously in this case. The memory mapped case does not suffer as badly as the large copy, as locality is better. The rump read/write case performs 10% better than the kernel due to a buffered backend. The tradeoff is increased memory use. In the unbuffered case the problem of not being able to execute a synchronous write operation while an asynchronous one is in progress shows.

Notably, we did not look into modifying the NetBSD kernel to provide better system call interfaces for selective cache flushing and I/O to character devices. For now, we maintain that performance for the typical workload is acceptable when compared to a kernel mount.

5 Related Work

The Alpine [9] network protocol development infrastructure provides an environment for running kernel code in userspace. It is implemented before the system call layer by overriding libc and is run in application process context. This approach both makes it unsuitable for statically linked programs and creates difficulties with shared global resources such as the `read()`/`write()` calls used for I/O beyond networking. Furthermore, from a file system perspective, this kind of approach shuts out kernel-initiated file system access, e.g. NFS servers.

Rialto [8] is an operating system with a unified interface both for userspace and the kernel making it possible to run most code in either environment. However, this system was designed from ground-up that way. Interesting ideas include the definition of both internal and

external linkage for an interface. While the ideas are inspiring, we do not have the luxury to redo everything.

Mach is capable of running Unix as a user process [11]. Lites [12] is a Mach server based on the 4.4BSD Lite code base. Debugging and developing 4.4BSD file systems under Mach/Lites is possible by using two Lites servers: one for the debugger and one for the file system being developed, including applications using the file system. If the Lites server being debugged crashes, applications inside it will be terminated. Being a single server solution, it does not provide isolation from the trusted computing base, either. A multiserver microkernel such as SawMill [10] addresses the drawbacks of a single server, but does not permit a monolithic mode or use of the file system code as an application library.

Operating systems running in userspace, such as User Mode Linux [7], make it possible to run the entire operating system as a userspace process. The main aims in this are providing better debugging & development support and isolation between instances. However, for development purposes, this approach does not provide isolation between the component under development and the core of the operating system - rather, they both run in the same process. This results in complexity in, for example, using fault injection and dynamic analysis tools. Neither does a userspace operating system integrate into the host, i.e. it is not possible to mount the userspace operating system as a file server. Even if that could be addressed, booting an entire kernel every time a ukfs application is run is a very heavyweight solution.

Sun's ZFS file system ships with a userspace testing library, libzpool [1]. In addition to kernel interface emulation routines, it consists of the Data Management Unit and Storage Pool Allocator components of ZFS compiled from the kernel sources. The ztest program plugs directly to these components. This approach has several shortcomings compared to rump file systems. First, it does not include the entire file system architecture, e.g. the VFS layer. The effort of implementing the VFS interface (in ZFS terms the *ZFS POSIX Layer*) was specifically listed as the hardest part of porting ZFS to FreeBSD [6]. Second, it does not facilitate userspace testing with real applications because it cannot be mounted. Third, the test program is specific to ZFS.

Many projects reimplement file system code for userspace purposes. Examples include e2fsprogs [30] and mtools [22]. Their implementation overlaps that which is readily already provided by the kernel. Especially e2fsprogs must track Linux kernel features and perform an independent reimplementation.

fuse-ext2 [2] is a userspace file server built on top of e2fsprogs. It implements a translator from FUSE to e2fsprogs. The functionality provided by fuse-ext2 is the same as that if rump_ext2fs, but requires specifi-

cally written code. The ChunkFS [13] prototype is fully mountable, but it is implemented on FUSE and userspace interfaces instead of the kernel interfaces.

Simulators [5, 29] can be used to run traces on file systems. Thekkath et al. [29] go as far as to run the HPUX FFS implementation in userspace. However, these tools execute against a recorded trace and do not permit mounting.

6 Conclusions and Future Work

In this paper we described the *Runnable Userspace Meta Program file system (rump fs)* method for using preexisting kernel file system code in userspace. There are two different modes of use for the framework: the p2k mode in which file systems are mounted so that they can be accessed transparently from any application, and a standalone mode in which applications can use file system routines through the ukfs library interface. The first mode brings a multiserver microkernel touch to a monolithic kernel Unix OS, but preserves a user option for monolithic operation. The second mode enables reuse of the available kernel code in applications such as those involved in image access. Implementations discussed in this paper were makefs and fs-utils.

The NetBSD implementation was evaluated. We discovered that rump file systems have security benefits especially with untrusted removable media. Rump file systems made debugging and developing kernel file system code easier and more convenient, and did not require additional case-specific “glue code” for making kernel code runnable in userspace. The issues regarding the maintenance of the rump shim were examined by looking at over a year’s worth of version control system commits. The build had broken 17 times and functionality 5 times. These were attributed to the lack of a full regression testing facility and developer awareness.

The performance of rump file systems using FFS was measured to be dependent of the type of backend. For file system images on a buffered regular file, properly synchronized performance was at best 10% faster than a kernel mount. Conversely, for an unbuffered character device backend the performance was at worst 40% slower. We attribute lower unbuffered performance to there being no standard interfaces for intermingling synchronous and asynchronous writes. We estimate typical workload performance to be $\pm 5\%$ of kernel mount performance. Future work may include optimizing performance, although for now we are fully content with it.

As a concluding remark, the technology has shown real world use and having kernel file systems from major open source operating systems available as portable userspace components would vastly increase system cross-pollination and reduce the need for reimplementa-

tations. We encourage kernel programmers to not only think about code from the classical machine dependent/machine independent viewpoint, but also from the environment dependent/environment independent perspective to promote code reuse.

Acknowledgments

This work has been funded by the Finnish Cultural Foundation, The Research Foundation of Helsinki University of Technology and Google Summer of Code.

The author thanks the anonymous reviewers for their comments and Remzi H. Arpacı-Dusseau, the shepherd, for his guidance in improving the final paper.

A special thank you goes to Arnaud Ysmal for implementing fs-utils. Additionally, the author thanks Simon Burge, André Dolenc, Johannes Helander, Luke Mewburn, Heikki Saikkonen, Chuck Silvers, Bill Stouder-Studenmund, Valeriy E. Ushakov and David Young for ideas, conversations, inspiring questions and answers.

Availability

The source code described in this paper is available for use and examination under the BSD license from the NetBSD source repository in the directory `src/sys/rump`. See <http://www.NetBSD.org/> for more information on how to obtain the source code.

References

- [1] ZFS source tour. [http://www.opensolaris.org/os /community/zfs/source/](http://www.opensolaris.org/os/community/zfs/source/).
- [2] AKCAN, A. fuse-ext2. <http://sourceforge.net/projects/fuse-ext2/>.
- [3] ALMEIDA, D. FIFS: a framework for implementing user-mode file systems in windows NT. In *WINSYM'99: Proc. of USENIX Windows NT Symposium* (1999).
- [4] BIANCUZZI, F. Interview about NetBSD WAPBL. *BSD Magazine* 2, 1 (2009).
- [5] BOSCH, P., AND MULLENDER, S. J. Cut-and-paste file-systems: Integrating simulators and file-systems. In *Proc. of USENIX* (1996), pp. 307–318.
- [6] DAWIDEK, P. J. Porting the ZFS file system to the FreeBSD operating system. In *Proc. of AsiaBSDCon* (2007), pp. 97–103.
- [7] DIKE, J. A user-mode port of the Linux kernel. In *ALS'00: Proc. of the 4th Annual Linux Showcase & Conference* (2000).
- [8] DRAVES, R., AND CUTSHALL, S. Unifying the user and kernel environments. Tech. Rep. MSR-TR-97-10, Microsoft, 1997.
- [9] ELY, D., SAVAGE, S., AND WETHERALL, D. Alpine: A User-Level infrastructure for network protocol development. In *Proc. of USITS'01* (2001), pp. 171–184.
- [10] GEFFLAUT, A., JAEGER, T., PARK, Y., LIEDTKE, J., ELPHINSTONE, K. J., UHLIG, V., TIDSWELL, J. E., DELLER, L., AND REUTHER, L. The sawmill multiserver approach. In *Proc. of the 9th ACM SIGOPS Europ. workshop* (2000), pp. 109–114.
- [11] GOLUB, D. B., DEAN, R. W., FORIN, A., AND RASHID, R. F. UNIX as an application program. In *Proc. of USENIX Summer* (1990), pp. 87–95.
- [12] HELANDER, J. Unix under Mach: The Lites server. Master's thesis, Helsinki University of Technology, 1994.
- [13] HENSON, V., VAN DE VEN, A., GUD, A., AND BROWN, Z. ChunkFS: using divide-and-conquer to improve file system reliability and repair. In *Proc. of HOTDEP'06* (2006).
- [14] HSUEH, M.-C., TSAI, T. K., AND IYER, R. K. Fault injection techniques and tools. *IEEE Computer* 30, 4 (1997), 75–82.
- [15] KANTEE, A. puffs - Pass-to-Userspace Framework File System. In *Proc. of AsiaBSDCon* (2007), pp. 29–42.
- [16] KANTEE, A., AND CROOKS, A. ReFUSE: Userspace FUSE Reimplementation Using puffs. In *EuroBSDCon 2007* (2007).
- [17] KERNIGHAN, B. Code testing and its role in teaching. *:login: The USENIX Magazine* 31, 2 (Apr. 2006), 9–18.
- [18] KLEIMAN, S. R. Vnodes: An architecture for multiple file system types in sun UNIX. In *Proc. of USENIX* (1986), pp. 238–247.
- [19] MEWBURN, L., AND GREEN, M. build.sh: Cross-building NetBSD. In *Proc. of BSDCon* (2003), pp. 47–56.
- [20] NETBSD PROJECT. <http://www.NetBSD.org/>.
- [21] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of PLDI* (2007), pp. 89–100.
- [22] NIEMI, D., AND KNAFF, A. Mtools, 2007. <http://mtools.linux.lu/>.
- [23] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON file systems. *SIGOPS OSR* 39, 5 (2005), 206–220.
- [24] SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *Proc. of USENIX Winter* (1993), pp. 307–326.
- [25] SILVERS, C. UBC: An efficient unified I/O and memory caching subsystem for NetBSD. In *Proc. of USENIX, FREENIX Track* (2000), pp. 285–290.
- [26] SMITH, K. A., AND SELTZER, M. I. File system aging—increasing the relevance of file system benchmarks. *SIGMETRICS Perform. Eval. Rev.* 25, 1 (1997), 203–213.
- [27] SNYDER, P. tmpfs: A virtual memory file system. In *Proc. EUUG Conference* (1990), pp. 241–248.
- [28] SZEREDI, M. Filesystem in USERSpace. <http://fuse.sourceforge.net/>.
- [29] THEKKATH, C. A., WILKES, J., AND LAZOWSKA, E. D. Techniques for file system simulation. *Software - Practice and Experience* 24, 11 (1994), 981–999.
- [30] TS’O, T. E2fsprogs: Ext2/3/4 Filesystem Utilities, 2008. <http://e2fsprogs.sourceforge.net/>.
- [31] WOODHOUSE, D. Jffs2 the journaled flash file system. In *Ottawa Linux Symposium* (2001).
- [32] YANG, J., SAR, C., TWOHEY, P., CADAR, C., AND ENGLER, D. Automatically generating malicious disks using symbolic execution. In *SP '06: Proc. of 2006 IEEE Symp. on Security and Privacy* (2006), IEEE Computer Society, pp. 243–257.
- [33] YSMAL, A. FS Utils. <http://NetBSD.org/~stackic/fs-utils.html>.
- [34] ZHANG, Z., AND GHOSE, K. hFS: a hybrid file system prototype for improving small file and metadata performance. In *Proc. of EuroSys* (2007), pp. 175–187.

Notes

¹ The kernel NFS server works in userspace, but is not yet part of the official source tree. There are conflicts between the RPC portmapper and mount protocol daemon for user- and kernel space nfs service. Basically, there is currently no way to differentiate if an exported directory hierarchy should be served by the kernel or userspace daemon.

² The NetBSD problem report database can be viewed with a web browser by accessing <http://gnats.NetBSD.org/<num>>, e.g. in the case of kern/38057 the URL is <http://gnats.NetBSD.org/38057>. The string “kern” stands for kernel and signifies the relevant subsystem.

CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems^{*}

Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat

{lohm, hofer, wosch}@cs.fau.de

FAU Erlangen–Nuremberg

Jochen Streicher, Olaf Spinczyk

{jochen.streicher, olaf.spinczyk}@tu-dortmund.de

TU Dortmund

Abstract

This paper evaluates aspect-oriented programming (AOP) as a first-class concept for implementing configurability in system software for resource-constrained embedded systems. To compete against proprietary special-purpose solutions, system software for this domain has to be highly configurable. Such fine-grained configurability is usually implemented “in-line” by means of the C preprocessor. However, this approach does not scale – it quickly leads to “#ifdef hell” and a bad separation of concerns. At the same time, the challenges of configurability are still increasing. AUTOSAR OS, the state-of-the-art operating-system standard from the domain of automotive embedded systems, requires configurability of even fundamental architectural system policies.

On the example of our CiAO operating-system family and the AUTOSAR-OS standard, we demonstrate that AOP – if applied from the very beginning – is a profound answer to these challenges. Our results show that a well-directed, pragmatic application of AOP leads to a much better separation of concerns than does #ifdef-based configuration – without compromising on resource consumption. The suggested approach of aspect-aware operating-system development facilitates providing even fundamental system policies as configurable features.

1 Introduction

The design and implementation of operating systems has always been challenging. Besides the sheer size and the inherent asynchronous and concurrent nature of operating-system code, developers have to deal with lots of crucial nonfunctional requirements such as performance, reliability, and maintainability. Therefore, researchers have always tried to exploit the latest advances in programming

languages and software engineering (such as object orientation [6], meta-object protocols [26], or virtual execution environments [14]) in order to reduce the complexity of operating system development and to improve the systems’ nonfunctional properties.

1.1 Operating Systems for Small Embedded Systems

This paper focuses on small (“deeply”) embedded systems. More than 98 percent of the worldwide annual production of microprocessors ends up in small embedded systems [24] – typically employed in products such as cars, appliances, or toys. Such embedded systems are subject to an enormous hardware-cost pressure. System software for this domain has to cope not only with strict resource constraints, but especially with a broad *variety* of application requirements and platforms. So to allow for reuse, an operating system for the embedded-systems domain has to be developed as a system-software product line that is highly configurable and tailorable. Furthermore, resource-saving static configuration mechanisms are strongly favored over dynamic (re-)configuration.

A good example for this class of highly configurable systems with small footprint is the new embedded operating-system standard specified by AUTOSAR, a consortium founded by all major players in the automotive industry [3]. The goal of AUTOSAR is to continue the success story of the OSEK-OS specification [19]. OSEK-compliant operating systems have been used in almost all European cars over the past ten years, which led to an enormous productivity gain in automotive software development. AUTOSAR extends the OSEK-OS specification in order to cover the whole system-software stack including communication services and a middleware layer.

Even in this restricted domain, there is already a huge variety of application requirements on operating systems. For instance, power-train applications are typically safety-critical and have to deal with real-time requirements,

*This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4, SCHR 603/7-1, and SP 968/2-1.

```

1 Cyg_Mutex::Cyg_Mutex() {
2     CYG_REPORT_FUNCTION();
3     locked      = false;
4     owner       = NULL;
5 #if defined(CYGSEM_PRI_INVERSION_PROTO_DEFAULT) && \
6     defined(CYGSEM_PRI_INVERSION_PROTO_DYNAMIC)
7 #ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_INHERIT
8     protocol    = INHERIT;
9 #endif
10 #ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_CEILING
11     protocol   = CEILING;
12     ceiling    = CYGSEM_PRI_INVERSION_PROTO_DEFAULT_PRI;
13 #endif
14 #ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_NONE
15     protocol   = NONE;
16 #endif
17 #else // not (DYNAMIC and DEFAULT defined)
18 #ifdef CYGSEM_PRI_INVERSION_PROTO_CEILING
19 #ifdef CYGSEM_DEFAULT_PRIORITY
20     ceiling = CYGSEM_DEFAULT_PRIORITY;
21 #else
22     ceiling = 0; // Otherwise set it to zero.
23 #endif
24 #endif
25 #endif // DYNAMIC and DEFAULT defined
26     CYG_REPORT_RETURN();
27 }

```

Figure 1: ”#ifdef hell” example from eCos [18]

while car body systems are far less critical. Hardware platforms range from 8-bit to 32-bit systems. Some applications require a task model with synchronization and communication primitives, whereas others are much simpler control loops. In order to reduce the number of electronic control units (up to 100 in modern cars [5]), some manufacturers have the requirement to run multiple applications on the same unit, which is only possible with guaranteed isolation; others do not have this requirement. To fulfill all these varying requirements, the AUTOSAR-OS specification [2] describes a family of systems defined by so-called *scalability classes*. It not only requires configurability of simple functional features, but also of all *policies* regarding temporal and spatial isolation. To achieve this within a single kernel implementation is challenging. The decision about fundamental operating-system policies (like the question if and how address-space protection boundaries should be enforced) is traditionally made in the early phases of operating-system development and is deeply reflected in its architecture, which in turn has an impact on many other parts of the kernel implementation. In AUTOSAR-OS systems, these decisions have to be postponed until the application developer configures the operating system.

1.2 The Price of Configurability

In a previous paper [17], we analyzed the implementation of static configurability in the popular eCos operating system [18], which also targets small embedded systems.

The system implements configurability in the familiar way with #ifdef-based conditional compilation (in C++). Even though eCos does not support configurability of architectural concerns as required by AUTOSAR (such as the memory or timing protection model), we have found an “#ifdef hell”, which illustrates that these techniques do not scale well enough. Maintainability and evolvability of the implementation suffer significantly. As an example, Figure 1 shows the “#ifdef hell” in the constructor of the eCos mutex class, caused by just four variants of the optional protocol for the prevention of priority inversion. However, the configurability of this protocol does not only affect the constructor code – a total of 34 #ifdef-blocks is spread over 17 functions and data structures in four implementation files.

As a solution, we proposed aspect-oriented programming (AOP) [15] and analyzed the code size and performance impact of applying AOP to factor out the scattered implementation of configurable eCos features into distinct modules called aspects.

1.3 Aspect-Oriented Programming

AOP describes a programming paradigm especially designed to tackle the implementation of *crosscutting concerns* – concerns that, even though conceptually distinct, overlap with the implementation of other concerns in the code by sharing the same functions or classes, such as the mutex configuration options in eCos.

In AOP, *aspects* encapsulate pieces of code called *advice* that implement a crosscutting concern as a distinct module. A piece of advice targets a number of *join points* (points in the static program structure or in the dynamic execution flow) described by a predicate called *pointcut expression*. Pointcut expressions are evaluated by the *aspect weaver*, which weaves the code from the advice bodies to the join points that are matched by the respective predicates.

As pointcuts are described declaratively, the target code itself does not have to be prepared or instrumented to be affected by aspects. Furthermore, the same aspect can affect various and even unforeseen parts of the target code. In the AOP literature [10], this is frequently referred to as the *obliviousness* and *quantification* properties of AOP.

The AOP language and weaver used in the eCos study and in the development of CiAO is AspectC++ [22], a source-to-source weaver that transforms AspectC++ sources to ISO C++ code, which can then be compiled by any standard-compliant C++ compiler.

Figure 2 illustrates the syntax of aspects written in AspectC++. The (excerpted) aspect *Priority_Ceiling* implements the priority ceiling variant of the eCos mutex class. For this purpose, it *introduces* a *slice* of additional elements (the member variable *ceiling*) into the

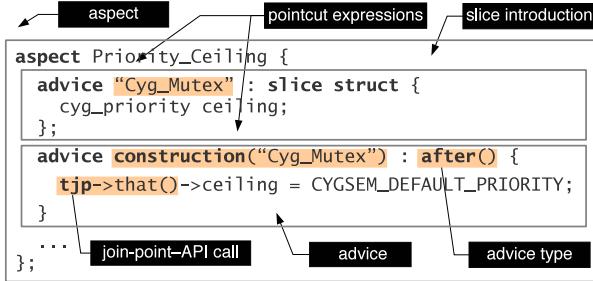


Figure 2: Syntactical elements of an aspect

class Cyg_Mutex and gives a piece of *advice* to initialize *ceiling* *after* each *construction* of a Cyg_Mutex instance. The targets of the introduction and the piece of construction advice are given by *pointcut expressions*.

In AspectC++, pointcut expressions are built from *match expressions* and *pointcut functions*. The match expression "Cyg_Mutex", for instance, returns a pointcut containing just the class Cyg_Mutex. Match expressions can also be fed into pointcut functions to yield pointcuts that represent events in the control flow of the running program, such as the event where some function is about to be *called* (*call()* advice) or an object instance is about to be *constructed* (see *construction("Cyg_Mutex")* in Figure 2). In most cases, the join points for a given pointcut can be derived statically by the aspect weaver so that the respective advice is also inserted statically at compile time without any run-time overhead.

The construction pointcut in the example is used to specify some *after* advice – that is, additional behavior to be triggered after the event occurrence. Other types of advice include *before* advice (speaks for itself) and *around* advice (replaces the original behavior associated with the event occurrence).

Inside the advice body, the type and pointer *JoinPoint* **tjp* provide an interface to the event context. The aspect developer can use this *join-point API* to retrieve (and partly modify) contextual information associated with the event, such as the arguments or return value of the intercepted function *call(tjp->arg(i), tjp->result())*. The *tjp->that()* in Figure 2 returns the *this* pointer of the affected object instance, which is used here to initialize the *ceiling* member variable (which in this case was introduced by the aspect itself).

1.4 Contribution and Outline

The results of applying AOP to eCos were very promising [17]. The refactored eCos system was much better structured than the original; the number of configuration points per feature could be drastically reduced. At the same time, we found that there is no negative impact on the system's performance or code size.

However, we also found that not all configurable features could be refactored into a modular aspect-oriented implementation. The main reason was that eCos did not expose enough unambiguous join points. We took this as a motivation to work on “aspect-aware operating system design”. This led to the development of fundamental design principles and the implementation of the CiAO¹ OS family for evaluation purposes. The idea was to build an operating system in an aspect-oriented way from scratch, considering AOP and its mechanisms *from the very beginning* of the development process. The resulting CiAO system is *aspect-aware* in the sense that it was analyzed, designed, and implemented with AOP principles in mind. In order to avoid evaluation results biased by the eCos implementation, CiAO was newly designed after the AUTOSAR-OS standard introduced above [2].

Our main goal is to evaluate the suitability of aspect-oriented software development as a first-class concept for the design and implementation of highly configurable embedded system software product-lines. The research contributions of this work are the following:

- Deeper insights on reasons for the #ifdef hell and the value of AOP in this context (Section 2).
- Design principles for aspect-aware operating system development (Section 4).
- CiAO: The first complete implementation of an operating system kernel developed with AOP concepts² (Section 5).
- A discussion of our results from CiAO (Section 6) and general experiences with the approach (Section 7).

For each of the topics, there is a dedicated section in the remaining part of this paper. In addition to that, Section 3 discusses relevant related work. The paper ends with our conclusions in Section 8.

2 Problem Analysis

Why exactly do state-of-the-art configurable systems like eCos exhibit badly modularized code termed as “#ifdef hell”? Is this an inherent property of highly configurable operating systems or just a matter of implementation means? In order to examine these questions, we took a detailed look at an abstract system specification, namely the AUTOSAR-OS standard introduced in Section 1.

¹CiAO is Aspect-Oriented

²The CiAO-OS family is freely available for research purposes from the authors.

	OS control	Tasks	ISRs category 1	ISRs category 2	Resources	Events	Alarms	Callbacks	Timing protection	Protection facilities	Foreign OS objects	Internal	
								Hooks	...	Invalid parameters	Wrong context	Interrupts disabled	...
... <3 OS services>	⊕							●	...	●	●	●	...
ActivateTask()	⊕							●	...	●	●	●	...
TerminateTask()	⊕							●	...	●	●	●	...
Schedule()	⊕							●	...	●	●	●	...
... <3 more task services>	⊕							●	...	●	●	●	...
ResumeAllInterrupts()	⊕							...	●	●
SuspendAllInterrupts()	⊕							...	●	●
... <7 more ISR services>	⊕	⊕						●	...	●	●	●	...
GetResource()	⊕							●	...	●	●	●	...
① ReleaseResource()	⊕							●	...	●	●	●	...
... <4 event services>	⊕							●	...	●	●	●	...
... <6 alarm services>	⊕							●	...	●	●	●	...
... <7 schedule table services>	⊕							●	...	●	●	●	...
... <7 OS application services>	⊕							●	...	●	●	●	...
② TaskType	⊕	⊗	⊗					...	⊗		⊗	...	⊗
ResourceType			⊕					...			⊗
... <4 more structures>	⊕	⊗	⊕	⊗	⊕			...	⊗		⊗
System startup	●		●					●
Task switch	●		●					●
Protection violation	●		●					●	...	●
... <4 more internal points>	●		●					●	...	●	...	●	...

Table 1: Influence of configurable concerns (columns) on system services, system types, and internal events (rows) in AUTOSAR OS [2, 19]; kind of influence: \oplus = extension of the API by a service or type, \otimes = extension of an existing type, \bullet = modification after service or event, \circlearrowleft = modification before, $\bullet\circlearrowleft$ = modification before and after

2.1 Why #ifdef Hell Appears to Be Unavoidable

The AUTOSAR-OS standard proposes a set of *scalability classes* for the purpose of system tailoring. These classes are, however, relatively coarse-grained (there are only four of them) and do not clearly separate between conceptually distinct concerns. CiAO provides a much better granularity; each AUTOSAR-OS concern is represented as an individual feature in CiAO, subject to application-dependent configuration.

In order to be able to grasp all concerns and their interactions, we have developed a specialized analysis method termed *concern impact analysis* (CIA) [13]. The idea behind CIA is to consider requirement documents together with domain-expert knowledge to develop a matrix of concerns and their influences in an iterative way. In the analysis of the AUTOSAR-OS standard, CIA yielded a comprehensive matrix, which is excerpted in Table 1.

The rows show the AUTOSAR OS system services (API functions) and system abstractions (types) in groups that represent distinct features. AUTOSAR OS is a statically configured operating system with static task priorities; hence, at run time, only services that alter the status of a *task* (e.g., setting it ready or suspended) are available. *Interrupt service routines* (ISRs), in contrast, are triggered asynchronously; the corresponding system functionality allows the application to prohibit their occur-

rence collectively or on a per-source basis. AUTOSAR OS distinguishes between two categories of ISRs that are somewhat comparable to top halves and bottom halves in Linux: Category-1 ISRs are scheduled by the hardware only and must not interact with the kernel. Category-2 ISRs, in contrast, run under the control of the kernel and may invoke other AUTOSAR-OS services. The third type of control flows supported by the AUTOSAR-OS kernel are *hooks*. Hooks define a callback interface for applications to be notified about kernel-internal events, such as task switch events or error conditions (see Column 8 in Table 1).

Resources are the means for AUTOSAR applications to ensure mutual exclusion to synchronize concurrent access to data structures or hardware periphery. They are comparable to mutex objects in other operating systems. In order to avoid priority inversion and deadlocks, AUTOSAR prescribes a stack-based priority ceiling protocol, which adapts task priorities at run time. Hence, a task never blocks on `GetResource()`. The only way for application tasks to become blocked is by waiting for an AUTOSAR-OS *event*; another task or ISR that sets that event can unblock that task.

Alarms allow applications to take action after a specified period of time; a *schedule table* is an abstraction that encapsulates a series of alarms. Finally, tasks, ISRs, and data can be partitioned into *OS applications*, which define a spatial and temporal protection boundary to be enforced

by the operating system.

The table lists selected identified concerns of AUTOSAR OS (column headings) and how we can expect them to interact with the named entities of the specification (row headings); that is, the 44 system services (e.g., `ActivateTask()`) and the relevant system abstractions (e.g., `TaskType`) as specified in [19, 2]. Furthermore, the lower third lists how we can expect concerns to impact *system-internal* transitions, which are not visible in the system API that is specified by the standard. Table 1 thereby provides an overview of how we can expect AUTOSAR-OS concerns to crosscut with each other in the structural space (abstractions, services) and behavioral space (control flow events) of the implementation.

The comprehensive table shows that a system that is built according to that specification will *inherently* exhibit extensive crosscutting between its concern implementations, leading to code tangling (many different concerns implemented in a single implementation module) and scattering (distribution of a single concern implementation across multiple implementation artifacts). This is because services like `ReleaseResource()` (see Table 1, row ①) and types like `TaskType` (see Table 1, row ②), for instance, are affected by as many as nine different concerns! That means that these implementations will exhibit at least nine `#ifdef` blocks – in the ideal case that each concern can be encapsulated in a single block, completely independent of the other concerns (which is unrealistic, of course). In fact, there is not a single AUTOSAR-OS service that is influenced by only one concern, which means that a straight-forward implementation using the C preprocessor will have numerous `#ifdefs` in every implementation entity. Thus, “`#ifdef hell`” seems unavoidable for the class of special-purpose, tailorabile operating systems.

2.2 Why AOP Is a Promising Solution

There are several properties inherent in AOP that are promising with respect to overcoming the drawbacks in `#ifdef`-based configuration techniques that were detailed above.

First, AOP introduces a new kind of binding between modules. In traditional programming paradigms, the caller module P (event producer) knows and *has to know* the callee module C (event consumer); that is, its name and interface (see Figure 3.a):

```
void C::callee() {
    <additional feature>
}
void P::caller() {
...
    C::callee(); // has to know C to bind feature
}
```

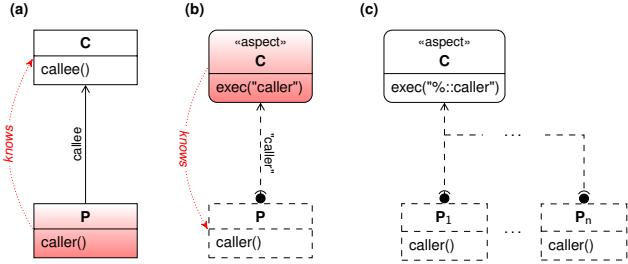


Figure 3: The mechanisms offered by AOP: advice-based binding and implicit per-join-point instantiation of advice

The advice-based binding mechanism offered by AOP can effectively invert that relationship: The callee (i.e., the aspect module C) can *integrate itself* into the caller (i.e., the base code P) without the caller having to know about the callee (see Figure 3.b):

```
advice execution("void P::caller()") : after() {
    <additional feature>
}
void P::caller() {
...
    // feature binds "itself"
}
```

If module C is optional and configurable, this loose coupling is an ideal mechanism for integration, because the call is implicit in the callee module. Using the traditional mechanisms, the call has to be included in the base module P and therefore has to be explicitly omitted if the feature implemented by module C is not in the current configuration. This configurable omission is realized by `#ifdefs` in state-of-the-art systems, bearing the significant disadvantages described above. A similar advantage of advice-based binding applies to configurable static program entities like classes or structures; aspects can integrate the state and operations needed to implement the corresponding feature into those entities *themselves* through slice introductions.

Second, by offering the mechanism of quantification through pointcut-expression matching, AOP allows for a modularized implementation of crosscutting concerns, which is also one of its main proclaimed purposes. This mechanism provides a flexible and implicit instantiation of additional implementation elements at compile-time (see Figure 3.c), ideally suited for the integration of concern implementations into configurable base code where the number of junction points (i.e., AOP join points) is flexible, ranging from zero to n :

```
advice execution("void %::caller()") : after() {
    <additional feature> // binds to any "caller()"
}
```

As we have seen in Table 1, most concerns in an AUTOSAR-OS implementation have a crosscutting im-

pact on many different points in the system in a similar way. An example is the policy that system services must not be called while interrupts are disabled (see Table 1, column ③). In the requirements specification of AUTOSAR OS, this policy is defined by requirement OS093:

If interrupts are disabled and any OS services, excluding the interrupt services, are called outside of hook routines, then the operating system shall return E_OS_DISABLEDINT. [2, p. 40]

This requirement can be translated almost “literally” to a single, modularized AspectC++ aspect:

```
aspect DisabledIntCheck {
    advice call(pcOSServices() && !pcISRServices())
    && !within(pcHooks()) : around() {
        if(interruptsDisabled())
            *tjp->result() = E_OS_DISABLEDINT;
        else
            tjp->proceed();
    } };
}
```

For convenience and the sake of separation of concerns, the aspect uses predefined *named pointcuts*, which are defined separately from the aspects in a global header file and specify which AUTOSAR-OS service belongs to which group:

```
pointcut pcOSServices() = "% ActivateTask()" || ...
pointcut pcISRServices() = ...
...
```

Using these named pointcuts, the aspect gives advice to all points in the system where any OS service but not the interrupt services are called:

```
call(pcOSServices() && !pcISRServices()) ...
```

The resulting set of join points is further filtered to exclude all events from within a hook routine:

```
... && !within(pcHooks())
```

Thus, we eventually get all calls outside of hook routines that are made to any service that is not an ISR service. The piece of around advice given to these join points performs a test whether the interrupts are currently disabled: If positive, the return code is set to the prescribed error code and the call is aborted; if negative, the call is performed as normal. (Around advice *replaces* the original processing of the intercepted event; however, it is possible to invoke the original processing explicitly with `tjp->proceed()`.)

The complete concern is encapsulated in this single aspect. The result is an enhanced separation of concerns in the system implementation. Layered, configurable systems can especially benefit from AOP mechanisms by being able to flexibly omit parts of the system without breaking caller–callee relationships.

3 Related Work

There are several other research projects that investigate the applicability of aspects in the context of operating systems. Among the first was the α -kernel project [7], in which the evolution of four scattered OS concern implementations (namely: prefetching, disk quotas, blocking, and page daemon activation) between versions 2 and 4 of the FreeBSD kernel is analyzed retroactively. The results show that an aspect-oriented implementation would have led to significantly *better evolvability* of these concerns. Around the same time, our own group experimented with AspectC++ in the PURE OS product line and later with aspect-refactoring eCos [17]. Our results from analyzing the AspectC++ implementation of various previously hard-wired crosscutting concerns show that this new paradigm leads to *no overhead* in terms of resource consumption per se.

Not a general-purpose AOP language but an AOP-inspired language of temporal logic is used in the Bossa project to integrate the Bossa scheduler framework into the Linux kernel [1]. Another example for a special-purpose AOP-inspired language is C4 [12, 21], which is intended for the application of kernel patches in Linux. The same goal of smarter patches (with a focus on “collateral evolutions” – changes to the kernel API that have to be caught up in dozens or hundreds of device drivers) is followed by Coccinelle [20]. Although the input language for the Coccinelle engine “SmPL” is not called an AOP language, it supports the modular implementation of crosscutting kernel modifications (i.e., quantification). Other related work concentrates on dynamic aspect weaving as a means for run-time adaptation of operating-system kernels: TOSKANA provides an infrastructure for the dynamic extension of the FreeBSD kernel by aspects [9]; KLASY is used for aspect-based dynamic instrumentation in Linux [25].

All these studies demonstrate that there are good cases for aspects in system software. However, both Bossa and our own work on eCos show that a useful application of AOP to existing operating systems requires additional AOP expressivity that results in run-time overheads (e.g., temporal logic or dynamic instrumentation). So far no study exists that analyzes the effects of using AOP for the development of an operating-system kernel from the very beginning. This paper explores just that.

4 Aspect-Aware Operating-System Development

The basic idea behind aspect-aware operating-system development is the strict separation of concerns in the *implementation*. Each implementation unit provides exactly one feature; its mere presence or absence in the config-

ured source tree decides on the inclusion of the particular feature into the resulting system variant.

Technically, this comes down to a strict decoupling of policies and mechanisms by using aspects as the primary composition technique: Kernel mechanisms are glued together and extended by aspects; they support aspects by ensuring that all relevant internal control-flow transitions are available as unambiguous and statically evaluable join points.

However, this availability cannot be taken for granted. Improving the configurability of eCos even further did not work as good as expected because of join-point ambiguity [17]. For instance, eCos does not expose a dedicated user API to invoke system services. This means that, on the join-point level, *user* \Rightarrow *kernel* transitions are not statically distinguishable from the kernel-internal activation and termination of system services. The consequence is that policy aspects that need to hook into these events become more expensive than necessary – for instance, an aspect that implements a new *kernel-stack* policy by switching stacks when entering/leaving the kernel. The ideal implementation of the kernel-stack feature had a performance overhead of 5% for the actual stack switches, whereas the aspect implementation induced a total overhead of 124% only because of unambiguous join points. The aspect had to use dynamic pointcut functions to disambiguate at run time: It used `cflow()`, a dynamic pointcut function that induces an extra internal control-flow counter that has to be incremented, decremented, and tested at run time to yield the join points. However, in other cases it was not possible at all to disambiguate, rendering an aspect-based implementation of new configuration options impossible.

We learned from this that the exposure of all relevant gluing and extension points as statically evaluable and unambiguous join points has to be understood as a primary design goal from the very beginning. The key premise for such *aspect awareness* is a component structure that makes it possible to influence the composition and shape of components as well as all run-time control flows that run through them by aspects [16].

4.1 Design Principles

The eCos experience led us to the three fundamental principles of aspect-aware operating-system development:

The principle of loose coupling. Make sure that aspects can hook into all facets of the static and dynamic integration of system components. The *binding* of components, but also their *instantiation* (e.g., placement in a certain memory region) and the time and order of their *initialization* should all be established (or at least be influenceable) by aspects.

The principle of visible transitions. Make sure that aspects can hook into all control flows that run through the system. All control-flow transitions into, out of, and within the system should be influenceable by aspects. For this they have to be represented on the join-point level as statically evaluable, unambiguous join points.

The principle of minimal extensions. Make sure that aspects can extend all features provided by the system on a fine granularity. System components and system abstractions should be fine-grained, sparse, and extensible by aspects.

Aspect awareness, as described by these principles, means that we moderate the AOP ideal of obliviousness, which is generally considered by the AOP community as a defining characteristic of AOP [11]. CiAO's system components and abstractions are *not* totally oblivious to aspects – they are supposed to provide explicit support for aspects and even depend on them for their integration.

4.2 Role and Types of Classes and Aspects

The relationship between aspects and classes is asymmetrical in most AOP languages: Aspects augment classes, but not vice versa. This gives rise to the question which features are best to be implemented as classes and which as aspects and how both should be applied to meet the above design principles.

The general rule we came up with in the development of CiAO is to provide some feature as a class if – and only if – it represents a *distinguishable instantiable concept* of the operating system. Provided as classes are:

1. **System components**, which are instantiated on behalf of the kernel and manage its run-time state (such as the Scheduler or the various hardware devices).
2. **System abstractions**, which are instantiated on behalf of the application and represent a system object (such as Task, Resource, or Event).

However, the classes for system components and system abstractions are sparse and to be further “filled” by *extension slices*. The main purpose of these classes is to provide a distinct scope with unambiguous join points for the aspects (that is, *visible transitions*).

All other features are implemented as aspects. During the development of CiAO we came up with three idiomatic roles of aspects:

1. **Extension aspects** add additional features to a system abstraction or component (*minimal extensions*), such as extending the scheduler by means for task synchronization (e.g., AUTOSAR-OS resources).

2. **Policy aspects** “glue” otherwise unrelated system abstractions or components together to implement some kernel policy (*loose coupling*), such as activating the scheduler from a periodic timer to implement time-triggered preemptive scheduling.
3. **Upcall aspects** bind behavior defined by higher layers to events produced in lower layers of the system, such as binding a driver function to interrupt events.

The effect of *extension aspects* typically becomes visible in the API of the affected system component or abstraction. *Policy aspects*, in contrast, lead to a different system behavior. We will see examples for extension and policy aspects in the following section. *Upcall aspects* do not contribute directly to a design principle, but have a more technical purpose: they exploit advice-based binding and the fact that AspectC++ inlines advice code at the respective join point for flexible, yet very efficient upcalls.

5 Case Study: CiAO-AS

CiAO is designed and implemented as a family of operating systems and has been developed from scratch using the principles of aspect-aware operating-system development. Note, however, that the application developer does not have to have any AOP expertise to use the OS. A concrete CiAO variant is configured statically by selecting features from a feature model in an Eclipse-based graphical configuration tool [4].

The *CiAO-AS* family member implements an operating-system kernel according to the AUTOSAR-OS standard³ [2], including configurable protection policies (memory protection, timing protection, service protection). The primary target platform for CiAO is the Infineon TriCore, an architecture of 32-bit microcontrollers that also serves as a reference platform for AUTOSAR and is widely used in the automotive industry.

5.1 Overview

Figure 4 shows the basic structure of the CiAO-AS kernel. Like most operating systems, CiAO is designed with a *layered architecture*, in which each layer is implemented using the functionality of the layers below. The only exceptions to this are the aspects implementing architectural policies, which may affect multiple layers.

On the coarse level, we have three layers. From bottom up these are: the *hardware access layer*, the *system layer* (the operating system itself), and the *API layer*.

³Because of legal issues, we do not claim full conformance; we have not performed any formal conformance testing.

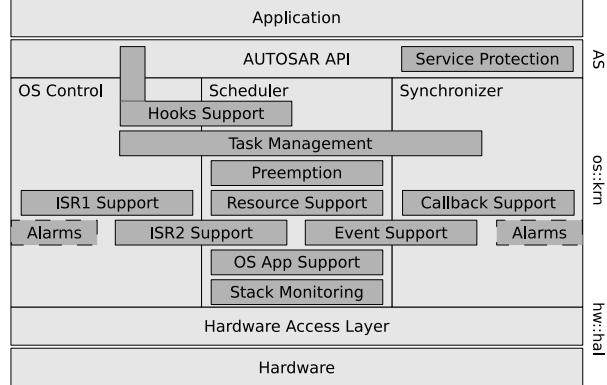


Figure 4: Structure of the CiAO-AS kernel

In CiAO, however, layers do not just serve conceptual purposes, but also are a means of aspect-aware development. With regard to the principle of *visible transitions*, each layer is represented as a separate C++ namespace in the implementation (hw::hal, os::krn, AS). Thereby, cross-layer control-flow transitions (especially into and out of os::krn) can be grasped by statically evaluable pointcut expressions. The following expression, for instance, yields all join points where a system-layer component accesses the hardware:

```
pointcut pc0S0toHW() = call("% hw::hal::%(...)")
&& within("% os::krn::%(...);");
```

5.2 The Kernel

In its full configuration, the system layer bears three logical *system components* (displayed as columns in Figure 4):

1. The *scheduler* (Scheduler) takes care of the dispatching of tasks and the scheduling strategy.
2. The *synchronization facility* (Synchronizer) takes care of the management of events, alarms, and the underlying (hardware / software) counters.
3. The *OS control facility* (OSControl) provides services for the controlled startup and shutdown of the system and the management of OSEK/AUTOSAR application modes.

However, as pointed out in Section 4.2, these classes are sparse or even empty. If at all, they implement only a minimal base of their respective concern. All further concerns and variants (depicted in dark grey in Figure 4) are brought into the system by aspects, most of which touch multiple system components and system abstractions.

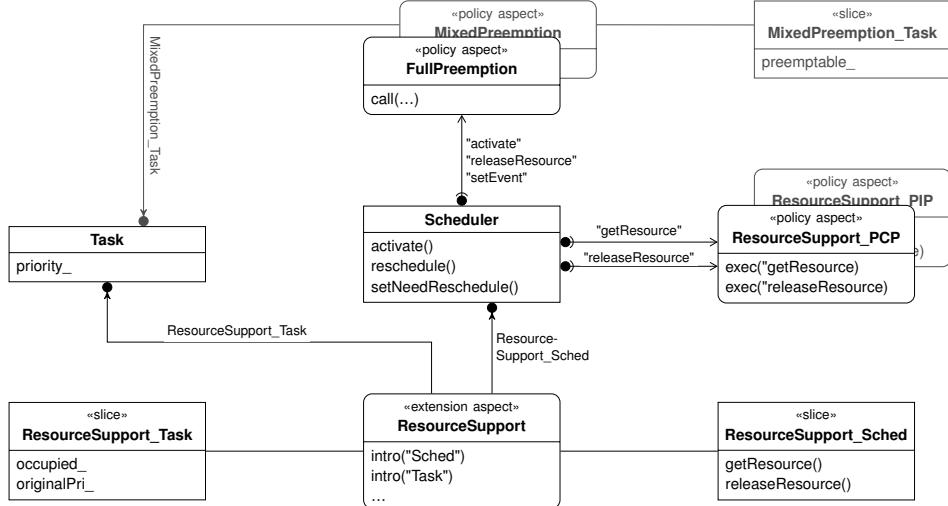


Figure 5: Interactions between optional policies and extensions of the CiAO scheduler

5.3 Aspect-Aware Development Applied

Figure 5 demonstrates how components, abstractions, and aspects engage with each other on a concrete example. The central element is the system component `Scheduler`. However, `Scheduler` provides only the minimal base of the scheduling facility, which is nonpreemptive scheduling:

```

class Sched {
    Tasklist ready_;
    Task::Id running_;
public:
    void activate(Task::Id whom);
    void reschedule();
    void setNeedReschedule();
    ...
};
    
```

Support for preemption and further abstractions is provided by additional *extension aspects* and *policy aspects*.

`ResourceSupport` is an example for an *extension aspect*. It extends the `Task` system abstraction `Scheduler` system component with support for resources. For this purpose, it introduces some state variables (`occupied_`, `originalPri_`) and operations (`getResource()`, `releaseResource()`).⁴ The elements to introduce are given by respective *extension slices*:

```

slice struct ResourceSupport_Task {
    ResourceMask occupied_;
    Pri originalPri_;
};
    
```

⁴ResourceSupport furthermore extends the API on the interface layer (it introduces the respective AUTOSAR-OS services `Get-/ReleaseResource()` and the `ResourceType` abstraction) so that applications can use the new functionality. For the sake of simplicity, this cross-layer extension is omitted here.

```

slice struct ResourceSupport_Sched {
    void getResource(Resource::Id resid) {...}
    void releaseResource(Resource::Id resid) {...}
};

aspect ResourceSupport {
    advice "Task" : slice ResourceSupport_Task;
    advice "Scheduler" : slice ResourceSupport_Sched;
};

FullPreemption is an example for a policy aspect. It implements the full-preemption policy as specified in [19], according to which every point where a higher-priority task may become ready is a potential point of preemption:
pointcut pcPreemptionPoints() =
    "% Scheduler::activate(...)" ||
    "% Scheduler::setEvent(...)" ||
    "% Scheduler::releaseResource(...)";

aspect FullPreemption {
    advice execution(pcPreemptionPoints()) : after() {
        tjp->that()->reschedule();
    }
};
    
```

The named pointcut `pcPreemptionPoints()` (defined in a global header file) specifies the potential preemption points. To these points, *if present*, the aspect `FullPreemption` binds the invocation of `reschedule()`. This demonstrates the benefits of *loose coupling* by the AOP mechanisms, which makes it easy to cope with conceptually different, but technically interacting features: In a fully-preemptive system without resource support, `Scheduler::releaseResource()` is just not present, thus does not constitute a join point for `FullPreemption`. However, if the `ResourceSupport` extension aspect is part of the current configuration, `Scheduler::release-`

concern	extension	policy	upcall	advice	join points	extension of advice-based binding to
ISR cat. 1 support	1	m	$2+m$	$2+m$		API, OS control m ISR bindings
ISR cat. 2 support	1	n	$5+n$	$5+n$		API, OS control, scheduler n ISR bindings
Resource support	1	1		3	5	scheduler, API, task PCP policy implementation
Resource tracking		1		3	4	task, ISR monitoring of Get/ReleaseResource
Event support	1			5	5	scheduler, API, task, alarm trigger action JP
Full preemption	1			2	6	3 points of rescheduling
Mixed preemption	1			3	7	task 3 points of rescheduling for task / ISR
Wrong context check	1			1	s	s service calls
Interrupts disabled check	1			1	30	all services except interrupt services
Invalid parameters check	1			1	25	services with an OS object parameter
Error hook			1	2	30	scheduler 29 services
Protection hook	1	1		2	2	API default policy implementation
Startup / shutdown hook			1	2	2	explicit hooks
Pre-task / post-task hook			1	2	2	explicit hooks

Table 2: Selected CiAO-AS kernel concerns implemented as aspects with number of affected join points. Listed are selected kernel concerns that are implemented as *extension*, *policy*, or *upcall aspects*, together with the related pieces of *advice* (not including order advice), the affected number of *join points*, and a short explanation for the purpose of each join point (separated by “|” into *introductions of extension slices* | *advice-based binding*).

Resource() implicitly triggers the advice. The separation of policy invocation from mechanism implementation makes it easy to integrate additional features, such as the ResourceSupport_PCP aspect, which implements a stack-based priority ceiling protocol for resources. As AspectC++ inlines advice code at the matching join point, this flexibility does not cause overhead at run time.

6 Discussion of Results

By following the principles of aspect-aware operating system development, policies and mechanisms are cleanly separated in the CiAO implementation. This separation is a golden rule of system-software development, but in practice difficult to achieve. While on the design level it is usually possible to describe a policy in a well-separated manner from underlying mechanisms, the implementation often tends to be crosscutting. The reason is that many system policies, such as the preemption policy, not only depend on decisions but also on the specific points in the control flow where these decisions are made. Here, the modularization into aspects shows some clear advantages.

6.1 Modularization of the System

Table 2 displays an excerpt of the list of AUTOSAR-OS concerns that are implemented as aspects in CiAO-AS. The first three columns list for each concern the number of *extension*, *policy*, and *upcall aspects* that implement the concern. (The resource-support aspect and the protection-hook aspect have both an extension and a policy facet.)

An interesting point is the realization of synergies by means of AOP *quantification*. If for some concern the number of pieces of advice is lower than the number of affected join points, we have actually profited from the AOP concept of quantification by being able to reuse advice code over several join points. For 8 out of the 14 concerns listed in Table 2, this is the case.

The net amount of this profit depends on the type of concern and aspect. *Extension aspects* typically crosscut *heterogeneously* with the implementation of other concerns, which means that they have specific pieces of advice for specific join points. These kinds of advice do not leave much potential for synergies by quantification. *Policy aspects* on the other hand – especially those for architectural policies – tend to crosscut *homogeneously* with the implementation of other concerns, which means that a specific piece of advice targets many different join points at once. In these cases, quantification creates significant synergies.

For all concerns, however, the implementation is realized as a distinct set of aspect modules, thereby reaching complete encapsulation and separation of concerns. Thus, any given feature configuration demanded by the application developer can be fulfilled by only including the implementation entities belonging to that configuration in the configured source tree to be compiled.

6.2 Scalability of the System

Execution Time. The effects of the achieved configurability also become visible in the CPU overhead. Table 3 displays the execution times of the micro-benchmark sce-

narios⁵ (**a**) to (**j**) and the comprehensive application (**k**) on CiAO and a commercial OSEK implementation⁶. For each scenario, we first configured both systems to support the smallest possible set of features (*min* columns in Table 3). The differences between CiAO and OSEK are considerable: CiAO is noticeably faster in all test scenarios.

One reason for this is that CiAO provides a much better configurability (and thereby granularity) than OSEK. As the micro-benchmark scenarios utilize only subsets of the OSEK/AUTOSAR features, this has a significant effect on the resulting execution times. The smallest possible configurations of the commercial OSEK still contained a lot of unwanted functionality. The scheduler is synchronized with ISRs, for instance; however, most of the application scenarios do not include any ISRs that could possibly interrupt the kernel.

To judge these effects, we performed additional measurements with an “artificially enriched” version of CiAO that provides the same amount of unwanted functionality as OSEK (column *full* in Table 3). This reduces the performance differences; however, CiAO is still faster in six out of eleven test cases. This is most notable in test case (**k**), which is a comprehensive application that actually *uses* the full feature set.

Another reason for the relative advantage of CiAO is that OSEK’s internal thread-abstraction implementation is less efficient. This is mainly due to particularities of the TriCore platform, which renders standard context-switch implementations ported to that platform very inefficient. CiAO, however, has a highly configurable and adaptable thread abstraction, therefore not only providing for an upward tailorability (i.e., to the needs of the application), but also downward toward the deployment platform.

Memory Requirements. In embedded systems, tailorability is crucial – especially with respect to memory consumption, because RAM and ROM are typically limited to sizes of a few kilobytes. Since system software does not directly contribute to the business value of an embedded system, scalability is of particular importance here. Thus, we also investigated how the memory requirements of the CiAO-AS kernel scale up with the number of selected configurable features; the condensed results

⁵All variants were woven and compiled for the Infineon TriCore platform with AC++-1.0PRE3 and TRICORE-G++-3.4.3 using -O3 -fno-rtti -funit-at-a-time -ffunction-sections -Xlinker --gc-sections optimization flags. Memory numbers are retrieved byte-exact from the linker-map files. Run-time numbers are measured with a high-resolution hardware trace unit (Lauterbach PowerTrace TC1796).

⁶ProOSEK is the leading commercial implementation of the OSEK standard and part of the BMW and Audi/VW standard cores. We compare CiAO against ProOSEK since (1) AUTOSAR is a true superset of OSEK and (2) we do not yet have access to a complete AUTOSAR implementation.

test scenario	CiAO		OSEK
	min	full	min
(a) voluntary task switch	160	178	218
(b) forced task switch	108	127	280
(c) preemptive task switch	192	219	274
(d) system startup	194	194	399
(e) resource acquisition	19	56	54
(f) resource release	14	52	41
(g) resource release with preemption	240	326	294
(h) category 2 ISR latency	47	47	47
(i) event blocking with task switch	141	172	224
(j) event setting with preemption	194	232	201
(k) comprehensive application	748	748	1216

Table 3: Performance measurement results [clock ticks]

are depicted in Table 4. Listed are the deltas in code, data, and BSS section size per feature that is added to the CiAO base system.

Each Task object, for instance, takes 20 bytes of *data* for the kernel task context (priority, state, function, stack, interrupted flag) and 16 bytes (*bss*) for the underlying CiAO thread abstraction structure. Aspects from the implementation of other features, however, may extend the size of the kernel task context. Resource support, for instance, crosscuts with task management in the implementation of the Task structure, which it extends by 8 bytes to accommodate the occupied resources mask and the original priority.

The cost of several features does not simply induce a constant cost, but depends on the number of affected join points, which in turn can depend on the presence of *other* features, as explained in Section 5.3 with the example of full preemption and resource support. This effect underlines again the flexibility of loose coupling by advice-based binding.

7 Experiences with the Approach

The CiAO results show that the approach of aspect-aware operating-system development is both feasible and beneficial for the class of configurable embedded operating systems. The challenge was to implement a system in which almost everything is configurable. In the following, we describe our experience with the approach.

7.1 Extensibility

We are convinced that the three design principles of aspect-aware operating-system development (*loose coupling*, *visible transitions*, *minimal extensions*) also lead to an easy extensibility of the system for new, unanticipated features. While it is generally difficult to prove the soundness of an approach for unanticipated change, we have at least some evidence that our approach has clear benefits

feature	with feature or instance	text	data	bss
<i>Base system (OS control and tasks)</i>				
per task	+ func	+ 20	+ 16 + stack	0
per application mode	0	+ 4	0	0
ISR cat. 1 support	0	0	0	0
per ISR	+func	0	0	0
per disable-enable	+ 4	0	0	0
Resource support	+ 128	0	0	0
per resource	0	+ 4	0	0
per task	0	+ 8	0	0
Event support	+ 280	0	0	0
per task	0	+ 8	0	0
per alarm	0	+ 12	0	0
Full preemption	0	0	0	0
per join point	+ 12	0	0	0
Mixed preemption	0	0	0	0
per join point	+ 44	0	0	0
per task	0	+ 4	0	0
Wrong context check	0	0	0	0
per void join point	0	0	0	0
per StatusType join point	+ 8	0	0	0
Interrupts disabled check	0	0	0	0
per join point	+ 64	0	0	0
Invalid parameters check	0	0	0	0
per join point	+ 36	0	0	0
Error hook	0	0	+ 4	0
per join point	+ 54	0	0	0
Startup hook or shutdown hook	0	0	0	0
Pre-task hook or post-task hook	0	0	0	0

Table 4: Scalability of CiAO’s memory footprint. Listed are the increases in static memory demands [bytes] of selected configurable CiAO features.

here:

In a specific real-time application project that we implemented using CiAO, minimal and deterministic event-processing latencies were crucial. The underlying hardware platform was the Infineon TriCore, which actually is a heterogeneous multi-processor-system-on-chip that comes with an integrated peripheral control processor (PCP). This freely-programmable co-processor is able to handle interrupts independently of the main processor. We decided to extend CiAO in a way that the PCP pre-handles all hardware events (interrupts) in order to map them to activations of respective software tasks, thereby preventing the real-time problem of rate-monotonic priority inversion [8]. This way, the CPU is only interrupted when there actually is a control flow of a higher priority than the currently executing one ready to be dispatched.

This relatively complex and unanticipated extension could nevertheless be integrated into CiAO by a single *extension aspect*, which is shown in Figure 6. The PCP_Extension aspect is itself a *minimal extension*; its implementation profited especially from the fact that all other CiAO components are designed according to the principle of *visible transitions*. This ensures here that all relevant transitions of the CPU, such as when the kernel is entered or left (lines 9 and 14, respectively) or when

the running CPU task is about to be preempted (line 17), are available as statically evaluable and unambiguous join points to which the aspect can bind.

Note, that the aspect in Figure 6 is basically the complete code for that extension, except for some initialization code (10 lines of code) and the PCP code, which is written in assembly language due to the lack of a C/C++ compiler for the PCP instruction set.

7.2 The Role of Language

We think that the expressiveness of the base language (in our case C++) plays an important role for the effectiveness of the approach. Thanks to modularization through namespaces and classes, C++ has some clear advantages over C with respect to *visible transitions*: the more of the base program’s purpose and semantics is expressed in its syntactic structure, the more unambiguous and “semantically rich” join points are available to which the aspects can bind.

Note, however, that even though CiAO is using C++, it is not developed in an object-oriented manner. We used C++ as a purely static language and stayed away from any language feature that induces a run-time overhead, such as virtual functions, exceptions, run-time type information, and automatic construction of global variables.

7.3 Technical Issues

Aspects for Low-Level Code. A recurring challenge in the development of CiAO was that the implementation of fundamental low-level OS abstractions, such as interrupt handlers or the thread dispatcher, requires more control over the resulting machine code than is guaranteed by the semantics of ISO C++. Such functions are typically (1) written entirely in external assembly files or (2) use a mixture of inline assembly and nonstandard language extensions (such as `__attribute__((interrupt))` in gcc). For the sake of *visible transitions*, we generally opted for (2). However, the resulting join points often have to be considered as fragile – if advice is given to, for instance, the context switch function, the transformations performed by the aspect weaver might break the programmer’s implicit assumptions about register usage or the stack layout. The workaround we came up with for these cases is to provide *explicit join points* to which the aspects can bind instead. Technically, an explicit join point is represented by an empty inline function that is invoked from the fragile code when the execution context is safe. CiAO’s context switch functionality, for instance, exposes four explicit join points to which aspects can bind: `before_CPURelease()`, `before_LastCPURelease()`, `after_CPUReceive()`, and `after_FirstCPUReceive()`. Because of function inlin-

```

1 aspect PCP_Extension {
2   advice execution("void hw::init()") : after() {
3     PCP::init();
4   }
5   advice execution("% Scheduler::setRunning(...)") :
6   before() {
7     PCP::setPrio(os::krn::Task::getPri(tjp->args<0>()));
8   }
9   advice execution("% enterKernel(...)") : after() {
10   // wait until PCP has left kernel (Peterson)
11   PCP_FLAG0 = 1; PCP_TURN = 1;
12   while ((PCP_FLAG1 == 1) && (PCP_TURN == 1)) {}
13 }
14 advice execution("% leaveKernel(...)") : before() {
15   PCP_FLAG0 = 0;
16 }
17 advice execution("% AST0::ast(...)") : around() {
18   // AST0::ast() is the AST handler that activates
19   // the scheduler (bound by an upcall aspect)
20
21   // wait until PCP has left kernel (Peterson)
22   PCP_FLAG0 = 1; PCP_TURN = 1;
23   while ((PCP_FLAG1 == 1) && (PCP_TURN == 1)) {}
24
25   // proceed to aspect that activates scheduler
26   tjp->proceed();
27   PCP_FLAG0 = 0;
28 }
29 advice execution("% Scheduler::schedule(...)") : after() {
30   // write priority of running task to PCP memory
31   PCP::setPrio(Task::getPri(
32     Scheduler::Inst().getRunning()));
33 }
34 };

```

Figure 6: PCP co-processor extension aspect

ing, this does not induce an overhead and the aspect code is still embedded directly into the context switch functionality.

Aspect–Aspect Interdependencies. In several cases we had to deal with subtle interdependencies between aspects that affect the same join points. For instance, the following aspect implements the *ErrorHook* feature, which exempts the application developer from manually testing the result code of OS services:

```

aspect ErrorHook {
  advice execution(pcOSServices() ... ) : after() {
    if(*tjp->result() != E_OK)
      invokeErrorHook(*tjp->result());
  }
}

```

Later we figured that, depending on the configuration, there are also other *aspects* that modify the result code. To fulfill its specification, *ErrorHook* has to be invoked after these other aspects. Whereas detecting such interdependencies was sometimes tricky (especially those that emerge only in certain configurations), they were generally easy to resolve by *order* advice:

```

advice execution(pcOSServices() ... ) : order(
  "ErrorHook", !"ErrorHook");

```

This type of advice allows the developer to define a (partial) order of aspect invocation for a pointcut. The precedence of aspects is specified as a sequence of match expressions, which are evaluated against all aspect identifiers. In the above example, the aspect yielded by the expression "*ErrorHook*" has precedence (is invoked *last* of all aspects that give *after* advice to the pointcut) over all other aspects (the result of !"*ErrorHook*"). Very helpful was that order advice does not necessarily have to be given by one of the affected aspects, instead it can be given by any aspect. This made it relatively easy to encapsulate and deal with configuration-dependent ordering constraints.

Join-Point Traceability. An important factor for the development were effective tools for join-point traceability. From the viewpoint of an aspect developer, the set of join points offered by some class implementation constitutes an interface. However, these interfaces are “implicit at best” [23]; a simple refactoring, such as renaming a method, might silently change the set of join points and thereby break some aspect. To prevent such situations, we used the Eclipse-based AspectC++ Development Toolkit (ACDT⁷), which provides a join-point-set delta analysis (very helpful after updating from the repository) and visualizes code that is affected by aspects. Thereby, unwanted side effects of code changes could be detected relatively easy.

8 Summary and Conclusions

Operating systems for the domain of resource-constrained embedded systems have to be highly configurable. Typically, such configurability is implemented “in line” by means of the C preprocessor. However, due to feature interdependencies and the fact that system policies and system mechanisms tend to crosscut with each other in the implementation, this approach leads to “#ifdef hell” and a bad separation of concerns. Our analysis of the AUTOSAR-OS specification revealed that these effects can already be found in the requirements; they are an *inherent phenomenon* of complex systems. If fundamental architectural policies have to be provided as configurable features, “#ifdef hell” appears to be unavoidable.

We showed that a pragmatic application of aspect-oriented programming (AOP) provides means for solving these issues: The advice mechanism of AOP effectively reverses the direction of feature integration; an (optional) feature that is implemented as an aspect *integrates itself* into the base code. Thanks to AOP’s pointcut expressions, the integration of features through join points is declarative – it scales implicitly with the presence or absence of

⁷<http://acdt.aspectc.org/>

other features. A key prerequisite is, however, that the system’s implementation exhibits enough unambiguous and statically evaluable join points. This is achieved by the three design principles of *aspect-aware operating-system development*.

By following this design approach in the development of CiAO, we did not only achieve the complete separation of concerns in the code, but also excellent configurability and scalability in the resulting system. We hope that our results encourage developers who start from scratch with a piece of configurable system software to follow the guidelines described in this paper.

Acknowledgments

We wish to thank the anonymous reviewers for EuroSys and USENIX for their helpful comments. Special thanks go to Robert Grimm, whose demanding and encouraging shepherding helped us tremendously to improve content and clarity of this paper.

References

- [1] ÅBERG, R. A., LAWALL, J. L., SÜDHOLT, M., MULLER, G., AND MEUR, A.-F. L. On the automatic evolution of an OS kernel using temporal logic and AOP. In *18th IEEE Int. Conf. on Automated Software Engineering (ASE ’03)* (Montreal, Canada, Mar. 2003), IEEE, pp. 196–204.
- [2] AUTOSAR. Specification of operating system (version 2.0.1). Tech. rep., Automotive Open System Architecture GbR, June 2006.
- [3] AUTOSAR homepage. <http://www.autosar.org/>, visited 2009-03-26.
- [4] BEUCHE, D. Variant management with pure::variants. Tech. rep., pure-systems GmbH, 2006. <http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf>, visited 2009-03-26.
- [5] BROY, M. Challenges in automotive software engineering. In *28th Int. Conf. on Software Engineering (ICSE ’06)* (New York, NY, USA, 2006), ACM, pp. 33–42.
- [6] CAMPBELL, R., ISLAM, N., MADANY, P., AND RAILA, D. Designing and implementing Choices: An object-oriented system in C++. *CACM* 36, 9 (1993).
- [7] COADY, Y., AND KICZALES, G. Back to the future: A retroactive study of aspect evolution in operating system code. In *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD ’03)* (Boston, MA, USA, Mar. 2003), M. Akşit, Ed., ACM, pp. 50–59.
- [8] DEL FOYO, L. E. L., MEJIA-ALVAREZ, P., AND DE NIZ, D. Predictable interrupt management for real time kernels over conventional PC hardware. In *12th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS ’06)* (Los Alamitos, CA, USA, 2006), IEEE, pp. 14–23.
- [9] ENGEL, M., AND FREISLEBEN, B. TOSKANA: a toolkit for operating system kernel aspects. In *Transactions on AOSD II* (2006), A. Rashid and M. Akşit, Eds., no. 4242 in LNCS, Springer, pp. 182–226.
- [10] FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKSIT, M. *Aspect-Oriented Software Development*. AW, 2005.
- [11] FILMAN, R. E., AND FRIEDMAN, D. P. Aspect-oriented programming is quantification and obliviousness. In *W’shop on Advanced SoC (OOPSLA ’00)* (Oct. 2000).
- [12] FIUCZYNSKI, M., GRIMM, R., COADY, Y., AND WALKER, D. patch(1) considered harmful. In *10th W’shop on Hot Topics in Operating Systems (HotOS ’05)* (2005), USENIX.
- [13] HOFER, W., LOHMANN, D., AND SCHRÖDER-PREIKSCHAT, W. Concern impact analysis in configurable system software—the AUTOSAR OS case. In *7th AOSD W’shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’08)* (New York, NY, USA, Mar. 2008), ACM, pp. 1–6.
- [14] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (2007), 37–49.
- [15] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *11th Eur. Conf. on OOP (ECOOP ’97)* (June 1997), M. Akşit and S. Matsuoka, Eds., vol. 1241 of *LNCS*, Springer, pp. 220–242.
- [16] LOHMANN, D. *Aspect Awareness in the Development of Configurable System Software*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2009.
- [17] LOHMANN, D., SCHELER, F., TARTLER, R., SPINCKY, O., AND SCHRÖDER-PREIKSCHAT, W. A quantitative analysis of aspects in the eCos kernel. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2006 (EuroSys ’06)* (New York, NY, USA, Apr. 2006), ACM, pp. 191–204.
- [18] MASSA, A. *Embedded Software Development with eCos*. New Riders, 2002.
- [19] OSEK/VDX GROUP. Operating system specification 2.2.3. Tech. rep., OSEK/VDX Group, Feb. 2005. <http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>, visited 2009-03-26.
- [20] PADIOLEAU, Y., LAWALL, J. L., MULLER, G., AND HANSEN, R. R. Documenting and automating collateral evolutions in Linux device drivers. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2008 (EuroSys ’08)* (Glasgow, Scotland, Mar. 2008).
- [21] REYNOLDS, A., FIUCZYNSKI, M. E., AND GRIMM, R. On the feasibility of an AOSD approach to Linux kernel extensions. In *7th AOSD W’shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’08)* (New York, NY, USA, Mar. 2008), ACM, pp. 1–7.
- [22] SPINCKY, O., AND LOHMANN, D. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20, 7 (2007), 636–651.
- [23] STEIMANN, F. The paradoxical success of aspect-oriented programming. In *21st ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA ’06)* (New York, NY, USA, 2006), ACM, pp. 481–497.
- [24] TURLEY, J. The two percent solution. *embedded.com* (Dec. 2002). <http://www.embedded.com/story/0EG20021217S0039>, visited 2009-03-26.
- [25] YANAGISAWA, Y., KOURAI, K., CHIBA, S., AND ISHIKAWA, R. A dynamic aspect-oriented system for OS kernels. In *6th Int. Conf. on Generative Programming and Component Engineering (GPCE ’06)* (New York, NY, USA, Oct. 2006), ACM, pp. 69–78.
- [26] YOKOTE, Y. The Apertos reflective operating system: the concept and its implementation. In *7th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA ’92)* (New York, NY, USA, 1992), ACM, pp. 414–434.

Automatically Generating Predicates and Solutions for Configuration Troubleshooting

Ya-Yunn Su

NEC Laboratories America

yysu@nec-labs.com

Jason Flinn

University of Michigan

jflinn@umich.edu

Abstract

Technical support contributes 17% of the total cost of ownership of today's desktop computers [12], and troubleshooting misconfigurations is a large part of technical support. For information systems, administrative expenses, made up almost entirely of people costs, represent 60–80% of the total cost of ownership [5]. Prior work [21, 24] has created promising tools that automate troubleshooting, thereby saving users time and money. However, these tools assume the existence of *predicates*, which are test cases painstakingly written by an expert. Since both experts and time are in short supply, obtaining predicates is difficult. In this paper, we propose a new method of creating predicates that infers predicates by observing the actions of ordinary users troubleshooting misconfigurations. We report on the results of a user study that evaluates our proposed method. The main results were: (1) our method inferred predicates for all configuration bugs studied with very few false positives, (2) using multiple traces improved results, and, surprisingly, (3) our method identified the correct *solutions* applied by users who fixed the misconfigurations in the study.

1 Introduction

Troubleshooting software misconfigurations is a difficult, frustrating job that consists of such tasks as editing configuration files, modifying file and directory access rights, checking software dependencies, and upgrading dynamic libraries. Troubleshooting is also costly: technical support represents 17% of the total cost of ownership of desktop computers [12], and administrative costs are 60–80% of the total cost of ownership of information systems [5].

Consequently, the research community has developed several systems that automate troubleshooting [21, 24]. However, these tools require *predicates*, which are test cases that evaluate the correctness of software configuration on a computer system. Unfortunately, creating predicates is currently a manual task that requires the participation of expert users with domain knowledge about the application being tested. Manual creation of predicates is time-consuming, and expert time is precious.

This paper shows how to substantially reduce the time and effort needed to deploy automated troubleshooting by *automatically* generating predicates from traces of ordinary users troubleshooting their computers. Our method is based on the insight that users who manually troubleshoot a misconfiguration generally use one or more commands to test the correctness of their system. If we can extract these commands and learn a function that maps command output to correct and incorrect configuration states, we can generate a predicate that can be used by an automated troubleshooting tool. Predicate extraction requires no extra effort on the part of the user who is troubleshooting a problem; it simply lets other users benefit from her efforts.

Our current prototype uses a modified Unix shell to observe user input and system output during troubleshooting. We chose a command line interface to simplify development and evaluation by limiting the types of input and output we need to observe. GUI applications pose additional challenges, and we discuss ways in which we might address those challenges in Section 7.1. We also use a modified Linux kernel to track the causal relationship between processes, files, and other kernel objects.

We use two heuristics to find commands that can be used as predicates. First, users generally test the system state multiple times while fixing a configuration problem: once to reproduce a configuration problem and possibly several other times to test potential solutions. Second, testing commands should generate some output that lets the user know whether or not the current configuration is correct. We therefore examine output consisting of data printed to the terminal, exit codes, and the set of kernel objects modified by a command execution. This output should be qualitatively different before and after a configuration problem is fixed. Thus, repeated commands with significant differences in output between two executions are flagged as predicates.

For example, a user could test whether a local Apache server is working correctly by using `wget` to retrieve the server's home page. When the user first runs the command, an HTTP error message is displayed. When

the user runs the command after fixing the problem, the home page is correctly displayed. Our method thus identifies the wget command as a predicate.

Our method also identifies potential solutions found during troubleshooting. We observe that, in general, a command that is part of a solution will causally affect the last execution of a predicate but not prior executions of the same predicate. Further, the output of the two executions of the predicate should differ because the problem is fixed during the latter execution but not during the former one. Thus, we use the causal information tracked by our modified kernel to select candidate solutions. We then sort our candidate solutions by the number of times they appear in the set of all traces. As with prior troubleshooting systems such as PeerPressure [22], we observe that the majority of users is usually right, so common solutions are more correct than uncommon ones.

We envision that our system can be deployed across a community of users of an application or an operating system. As users encounter and solve new configuration problems, our tool will collect traces of their activity, analyze the traces, and upload canonicalized and anonymized predicates and solutions to a centralized repository. Other people executing automated configuration management tools can then benefit from the experience of these prior users by downloading and trying commonly used predicates and solutions. Note that since tools such as AutoBash have the ability to try multiple solutions and deterministically roll back unsuccessful ones, our solution database does not have to identify the exact solution to a problem, just the most likely ones. This kind of community sharing of experience has been previously proposed for use with tools that prevent deadlocks [11] and enable better understanding of software error messages [9].

We evaluated the effectiveness of our method by conducting a user study in which we recorded the actions of twelve people fixing four configuration problems for the CVS version control system and the Apache Web server. We then used our method to extract predicates from these recordings. Aggregating across all traces, our method found 22 predicates that correctly tested the state of these systems, while generating only 2 false positives. Further, our method was able to correctly identify the solution that fixed each misconfiguration.

2 Related work

The research community has proposed several different approaches to automate troubleshooting. Some systems, such as Chronus [24] and AutoBash [21], search through the space of possible configurations to find a correct one. Chronus searches backwards in time to find the instance in which a misconfiguration was introduced.

AutoBash applies actions previously taken to fix a problem on one computer to fix similar problems on other computers. Both systems use predicates that test software correctness to guide the search. Initially, one or more predicates evaluate to false, indicating a misconfiguration. Both systems use checkpoint and rollback to search for a state in which all predicates evaluate to true; a system in this state is assumed to be configured correctly. Using checkpoints that are created prior to executing predicates, both systems ensure predicates do not permanently affect the state of the system because execution is rolled back after the predicate completes.

Clearly, successful troubleshooting relies on having good predicates. Without predicates that distinguish correct and incorrect configurations, the automated search process will fail. Both Chronus and AutoBash previously assumed that experts would create such predicates by hand, which limits their applicability. In this paper, we show how predicates can be *automatically* generated by observing ordinary users troubleshooting configuration problems. We also show how we can simultaneously generate candidate solutions, which can be tried by systems such as AutoBash during the search.

Other automated troubleshooting tools, such as Strider [23] and PeerPressure [22], take a state-based approach in which the static configuration state on one computer is compared with that on other computers to identify significant deviations. Since a state-based approach does not need predicates, our work is not directly applicable to such systems. One of the principles guiding the design of these systems is that the majority of users (and, hence, the most frequent configuration states) are usually correct. We apply this same principle in our work to identify candidate solutions, and the results of our user study support this hypothesis.

Nagaraja et al. [16] observed that operator mistakes are an important factor in the unavailability of on-line services and conducted experiments with human operators. Like our user study, their experiments asked operators to perform maintenance tasks and troubleshoot misconfigurations for their three-tiered Internet service. However, their study did not share our goal of identifying predicates for automated troubleshooting.

The software testing research community faces a problem similar to ours, namely that manually writing test cases for software testing is tedious and time-consuming. Two solutions proposed by that community are automatic test case generation and test oracles. However, automatic test case generation [3, 4, 6, 8, 10, 13, 18] requires a specification for the program being tested. A test oracle determines whether a system behaves correctly for test execution. Researchers also have developed ways to automate test oracles for reactive systems from specifications [19, 20] and for GUI applications

using formal models [14]. Unfortunately, generating a specification or formal model requires substantial expert participation. Since the need for such expertise is exactly what we are trying to eliminate, both methods are inappropriate for our purpose, except when such specifications and models already have been created for another purpose.

Snitch [15] is a misconfiguration troubleshooting tool that analyzes application state to find possible root causes. It uses an always-on tracing environment that records application state changes and uses exit codes as *outcome markers* to identify faulty application traces. Our work differ from Snitch in two ways. First, our work generates test cases that verify system configurations, while Snitch finds root causes of misconfigurations. Since Snitch only records application state changes, its traces do not contain sufficient information to generate predicates. In contrast, our traces record all user commands; from these commands, we select which ones can be used as predicates. Second, Snitch only uses exit codes to analyze outcomes. Our work traces the causal relationship between commands executed by the user to better understand relationships among those commands. We also use semantic information from screen output to analyze the outcome of commands.

3 Design principles

We followed three goals in designing our automated predicate extraction system.

3.1 Minimize false positives

Automatically inferring user intentions is a hard problem. It is unreasonable to expect that we can perfectly identify potential predicates and solutions in every case. Thus, we must balance false positives (in which we extract predicates that incorrectly test configuration state) and false negatives (in which we fail to extract a correct predicate from a user trace).

In our domain, false positives are much worse than false negatives. A false positive creates an incorrect predicate that could potentially prevent the automated troubleshooting tool from finding a solution or, even worse, cause the tool to apply an incorrect solution. While a false negative may miss a potentially useful test case, we can afford to be conservative because we can aggregate the results of many troubleshooting traces. With multiple traces, a predicate missed in one trace may still be identified using another trace. Thus, in our design, we bias our system toward generating few false positives, even though this bias leads to a higher rate of false negatives.

3.2 Be as unobtrusive as possible

Troubleshooting is already a tedious, frustrating task. We do not want to make it even worse. If we require

users to do more work than they would normally need to do to troubleshoot their system, they will likely choose not to use our method. For instance, as described in the next section, we initially considered asking users to specify which commands they used as predicates or to specify rules that could be used to determine which commands are predicates. However, we eventually decided that these requirements were too burdensome.

Instead, our design requires only the most minimal involvement of the user. In particular, the user must start a shell that they will use to perform the troubleshooting. The shell does all the work necessary to record input, output, and causal dependencies. It then canonicalizes the trace to replace identifiers such as userids, host names, and home directories with generic identifiers. For example, the userid, yysu, would be replaced with USERID, and her home directory would be replaced with HOMEDIR. The traces can then be shipped to a centralized server for processing. The server might be run by a company’s IT department, or it might be an open-source repository.

As an added incentive, our shell provides functionality that is useful during troubleshooting, such as checkpoint and rollback [21]. However, this added functionality is not essential to the ideas in this paper.

3.3 Generate complete predicates

Users often test software configurations with complex, multi-step test cases. For example, to test the CVS repository, a user might import a test project, check it back out again to a new directory, and use the Unix diff tool to compare the new and old versions.

A predicate extraction method should identify all steps in multi-step predicates. For instance, if it only detected the latter two steps, the predicate it generates would be incorrect. Applying the predicate on another computer will always lead to predicate failure since that computer will not have the test project in the repository. Asking user to identify missing steps is intrusive, violating our previous design goal. Therefore, we use a different method, causal dependency tracking, to identify missing steps. Once we identify a repeated command that has different qualitative output, we identify all prior steps on which that command depends. We refer to such commands as *preconditions*, and we include them as part of the extracted predicate.

4 A failed approach and the lessons learned

In this section, we describe our first, unsuccessful approach to inferring predicates and the lessons we learned. Our initial idea was to record the actions of users troubleshooting misconfigurations and ask them to classify which of the commands they had entered were predicates. We hoped to use this data as a training set for

machine learning. Our goal was to develop a classifier that would correctly identify predicates from subsequent user traces. We hoped that a machine learning approach would meet our design goals by generating few false positives and not requiring any user feedback during normal operation (i.e., once the training set had been collected).

Unfortunately, this initial approach ran into two pitfalls: First, it was difficult for users to classify their own commands. Although we explained the concept of predicates to our users, it was still difficult for them to select predicates from a list of the commands that they entered. We found that users would often classify a command as a predicate only if it generated some output that was a recognizable error condition. Different users would classify the same command differently, and the same users would classify identical commands as being predicates in some cases but not in others. Thus, the training set was very noisy, making it difficult to use for machine learning.

Second, for many predicates identified by users, we found it difficult to determine which output represented success and which represented failure. For example, many users identified `ls -l` as a predicate because they would inspect a directory's contents to examine whether specific files were present with the appropriate permissions. To evaluate this predicate automatically, we would need to determine what the user expected to see in the directory. For instance, the user might be searching for a test project that he had just imported into CVS and verifying that it was not globally readable. For such commands, the evaluation of the predicate is extremely context-dependent.

These two pitfalls caused us to re-examine our approach. We observed that users often take an *action-based approach*, a *state-based approach*, or a combination of both to troubleshoot a configuration problem. An action-based approach means that the user interacts with the misconfigured application to learn the behavior of the application. For example, when troubleshooting a CVS problem, the user might `cvs import` a new module or `cvs checkout` a module. The user learns about the misconfiguration by examining the error and progress-reporting messages displayed on the screen. A state-based approach means that the user passively examines relevant state on which the application depends. For example, when troubleshooting the same CVS problem, the user would see if the CVS repository directory exists, what the CVS repository file permissions are, list the user and group information, etc.

Commands used in both action-based and state-based approaches can be used for testing. We observe that if we use commands used in an action-based approach as predicates, it is often easier to classify their return value. When the system state is incorrect, processes that execute these commands often have different exit values or

	Action-based commands	State-based commands
Test system state	<code>wget</code> <code>cvs</code>	<code>ls</code> <code>groups</code>
Do not test system state	<code>chmod</code> <code>usermod</code>	read config file clear terminal screen

This table shows examples of various commands that fall into one of four possible combinations. Our methodology finds commands in the shaded quadrant.

Table 1. Command taxonomy

display error messages. On the other hand, commands used in a state-based approach are often more difficult to evaluate because determining correctness requires semantic information or knowledge about the problem itself to reason about observable output.

This led us to develop the taxonomy in Table 1. The horizontal axis classifies commands according to how easy it is to determine their success or failure. On the left, we list commands that are action-based; these are easy to classify because they generate error messages, return different exit values, or modify a different set of objects when they fail. On the right, we list state-based commands such as `ls` that are hard to classify based on their output. The vertical axis classifies commands according to whether they are used to test system state.

We observed that only commands that fall in the top left quadrant are truly useful for automated troubleshooting. For a state-based command such as `ls`, the automated troubleshooting system will find it very hard to tell whether a difference in screen output is due to a misconfiguration or simply an inconsequential difference in system state (such as two users choosing different names for a test project).

This realization led us to a new approach. Rather than first try to identify predicates and then determine how to classify their output, we decided to instead *identify only repeated commands that have output that can be classified easily*. While this approach will miss some potential predicates, the omitted predicates are likely not useful for automated troubleshooting anyway because of the difficulty in classifying their output. Our prototype, described in the next section, uses this new approach.

5 Implementation

We first give an overview of our automated troubleshooting tool followed by our base algorithm, which we use to find single-step predicates.. All commands found by our base algorithm are action-based commands, as these commands exhibit output that our base algorithm could classify as success or failure. We then describe a refinement to the base algorithm that allows us to also identify multi-step predicates.

5.1 Overview

We envision that when a user or system administrator encounters a configuration problem, she will launch our troubleshooting shell to fix the problem. Our shell records all the commands she types and uses the algorithms described in Sections 5.2 and 5.3 to determine which commands are predicates and solutions. The user can then save the predicates and solutions for later and share them with an online repository. To help classify predicates and solutions, the user may identify the specific application she is troubleshooting.

Later, if another user runs into a configuration problem for the same application, he can download the solutions and predicates generated by other users. Our previous work, AutoBash [21], uses speculative execution to try potential solutions. After executing each solution, AutoBash tests if the system is working by executing predicates. If all predicates evaluate to true, AutoBash declares that the configuration problem is fixed and commits the solution execution. Operating system support for speculative execution [17] enables the safe roll back of state changes made by predicates and failed solutions.

5.2 Base algorithm

Our algorithm searches for repeated commands that differ in at least two out of the following output features:

- **A zero or non-zero exit value.** When a shell creates a process to execute a command, the return value of the process, typically specified using the `exit` system call, is returned to the shell. Conventionally, a Unix command returns a non-zero exit code to indicate failure and zero to indicate success. Our troubleshooting shell simply records the exit value for each command.
- **The presence of error messages in screen output.** Human beings are good at understanding screen output that gives feedback on the progress or result of executing a command. However, the screen output may contain unstructured text that is difficult for computers to analyze. Our hypothesis is that we only need to search for the presence of certain positive or negative keywords in the screen output to determine if a repeated command has different output. Searching for specific words is much simpler than trying to understand arbitrary screen output from the application and is often sufficient to know that two executions of a repeated command are different. Further, such semantic clues have been successfully applied in other domains to help search for programming bugs [7].

One alternative would be to say that two commands differ if there is any difference in their screen output. However, many commands such as `date` often

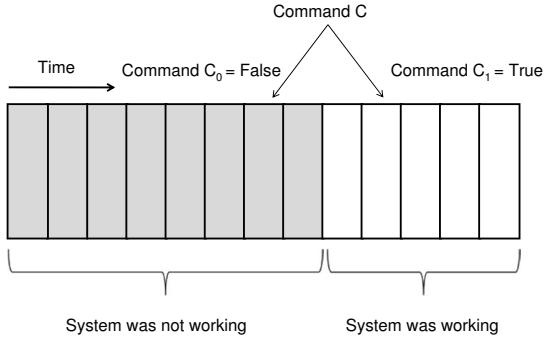
generate minor differences in output that do not reflect whether the command succeeded or failed.

Our shell intercepts all screen output displayed to the user and writes it to a log file. Our implementation is very similar to the Unix `script` tool in that it uses Unix's pseudo-terminal interface to log output transparently to the user. Our base algorithm asynchronously searches for semantic clues in the log file. Currently, it only searches for the presence of the word "error" or any system error messages as defined in the file `errno.h`; e.g., "Permission denied" or "No such file or directory." When *any* error message is found in the screen output, the command is considered to contain an error message. Otherwise, the command is classified as not containing an error message. Thus, two executions of the same command are considered to differ in this output feature if one generates an error message and the other does not.

- **The command's output set.** We define a command's *output set* to be the set of all files, file metadata, directories, directory entries, and processes modified by the execution of the command. Our shell uses our modified Linux kernel to trace the causal effects (output set) of command execution. Initially, only the process executing a command is in the output set. Our kernel intercepts system calls to observe the interaction of processes with kernel objects such as files, directories, other processes, pipes, UNIX sockets, signals, and other forms of IPC. When such objects are modified by the command's execution, they are added to the output set. When another process interacts with an object already in the output set, that process is added to the output set. For instance, if a command forks a child process that modifies a file, both the child and file are added to the output set. If another process reads the file, that process is also added to the output set. Transient objects, such as temporary files created and deleted during command execution, are not considered part of the command's output set.

Our current implementation tracks the output set for each command line. Therefore, when a user executes a task with a sequence of commands connected by a pipe, the troubleshooting shell creates only one output set. We chose this implementation because commands connected by a pipe are causally related by the pipe, so their output sets are usually the same.

Recently, our research group developed an alternative method for tracking output sets that does not require using a modified kernel [1]. This method uses system call tracing tools such as `strace` to



This figure shows a command C that is executed twice. The first time is shown as C_0 , and the second is shown as C_1 . If C_0 and C_1 have different output features, C_0 would return false and C_1 would return true.

Figure 1. The insight behind the base algorithm

generate the output set in a manner similar to that described above. In the future, we plan to use this alternative method to eliminate the dependency on kernel modifications.

Two executions of the same command are considered to have different output sets if any object is a member of one output set but not the other. This comparison does not include the specific modifications made to the object. For instance, the output sets of two executions of the command `touch foo` are equivalent because they both contain the metadata of file `foo`, even though the two executions change the modification time to different values.

Figure 1 shows the insight behind the base algorithm. If a repeated command has two or more different output features when executed at two different points in time, it is likely that the command is a predicate being used to test system state. We hypothesize that users are likely to create a test case to demonstrate a problem, attempt to fix the problem, and then re-execute the test case to see if the fix worked. If this hypothesis is true, then the last execution of the repeated command should represent a successful test and have recognizably different output than prior, unsuccessful tests. This troubleshooting behavior was previously reported in computer-controlled manufacturing systems [2] and happened in 28 out of 46 traces in our user study described in Section 6.

Based on this observation, we say that a predicate evaluates to true (showing a correct configuration) if two or more output features match those in the final execution of the command. If two or more differ, we say the predicate evaluates to false and shows a misconfiguration. Note that some predicates that return a non-zero exit code and generate an error message should still evaluate to true. For instance, a user may be testing whether an unauthorized user is able to access files in a CVS

repository. In this case, an error message actually indicates that the system is configured correctly.

Identifying repeated commands as the same command based on strict string comparison is somewhat problematical because it may leave out critical information. First, a command may be executed as different users; e.g., once as an ordinary user and once as root. This can lead to substantially different output even if a misconfiguration was not corrected between the two executions. Second, the same command may be executed in two different working directories, leading to markedly different output.

We solved this problem by having our shell record the userid and working directory for each command. If either of these variables differ, we consider two commands to be different. This is a conservative approach; e.g., the working directory has no effect on the value returned by `date`. While our conservative approach may miss good predicates, it is consistent with our design goal of preferring false negatives to false positives.

Some known environment variables, such as the parent process id, are ignored by our shell because we have judged them extremely unlikely to influence command execution. However, unknown environment variables introduced during troubleshooting are always recorded and used to differentiate commands. It may be that further experience will allow us to refine our heuristic for which environment variables to exclude. However, our evaluation in Section 6 shows that comparing only the userid, working directory, and any new environment variables introduced during troubleshooting generated many correct predicates. Further, no false positives were generated due to environment variables excluded from the comparison.

5.3 Finding preconditions

The base algorithm is very effective in identifying single-step predicates that test system state. However, we noticed that in some cases a predicate relies on some prior commands to change the system state before it can work correctly. We call these prior commands on which a predicate depends *preconditions*. A predicate that relies on preconditions cannot be used without those preconditions by automated troubleshooting tools as it would always return the same result. For example, if a user fixed a CVS problem starting from an empty repository and the base algorithm identified `cvs checkout` to be a predicate, this predicate would work only if the command `cvs import` had first been executed. The `cvs import` command is a precondition for that predicate because, without it, the predicate will always return false.

To find preconditions, we first search for commands that have causal effects on a predicate. We identify causal relationships by tracking the output set of each command, even after that command terminates. If a sub-

sequent command becomes a member of a prior command's output set, then we say that the prior command has a causal effect on the latter one. For instance, the command `echo hi > foo` would have a causal effect on the subsequent command `cat foo`.

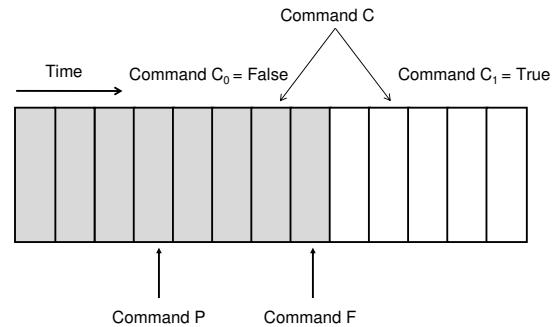
Besides including prior commands that have causal effects on predicates, we also include prior commands that add or remove environment variables. Since the base algorithm compares environment variables when comparing commands, adding or removing environment variables is considered to have an effect on all subsequent commands. This is more conservative than strictly needed, as environment variables could be added but not read by some or all later commands executed in the same shell. However, identifying such situations requires application-specific semantic knowledge.

Not all prior commands with causal effects on predicates are preconditions. If the user has successfully fixed the configuration problem, the commands comprising a solution will have a causal effect on the last execution of the predicate. Including a solution as a precondition would cause a predicate to always return true, rendering the predicate ineffective. Note that even if a solution embedded in a predicate inadvertently fixes a problem, the effects are undone when the automated troubleshooting system rolls back system state after predicate execution.

We differentiate between preconditions and solutions by first finding all commands that causally affect all executions of a predicate. Within these commands, the heuristic uses two rules to determine if a command is a precondition or a solution. First, a command that has causal effects on both successful and failed predicates is a precondition. Second, a command that only has causal effects on successful predicates and is executed chronologically after all failed predicates is a solution.

Figure 2 shows the insight that allows the second rule to distinguish between a solution and a precondition. The insight is that users will usually first make sure preconditions are executed before trying to run a test case that reproduces a misconfiguration (since the test can not succeed without the precondition). Therefore, commands that occur after failed predicates and before successful ones are likely to be solutions.

Consider an example in which both `user1` and `user2` are authorized to access a CVS repository, but only `user1` is in the CVS group. Figure 3 shows the four commands the user executes and the causal relationship between commands. First, the base algorithm would determine that "`cvs co as user2`" is a predicate. The first predicate execution is considered to return false and the second one is considered to return true. Both the "`cvs import as user1`" command and the "`usermod -G CVSgroup user2`" command, which adds `user2` to CVSgroup, have causal effects on the second predicate



This figure demonstrates how the precondition heuristic works. Assume the base algorithm determines that C_0 returns false and C_1 returns true. If both a command P and a command F have causal effects on C_1 only, the precondition heuristic determines that the command P is a precondition for command C and the command F is a solution. The reason is that command F is likely the solution that causes the state transition that causes C_0 and C_1 to return different results.

Figure 2. The insight behind the precondition heuristic

execution. The precondition uses the chronological order to determine that "`cvs import as user1`" is a precondition and "`usermod -G CVSgroup user2`" is a solution.

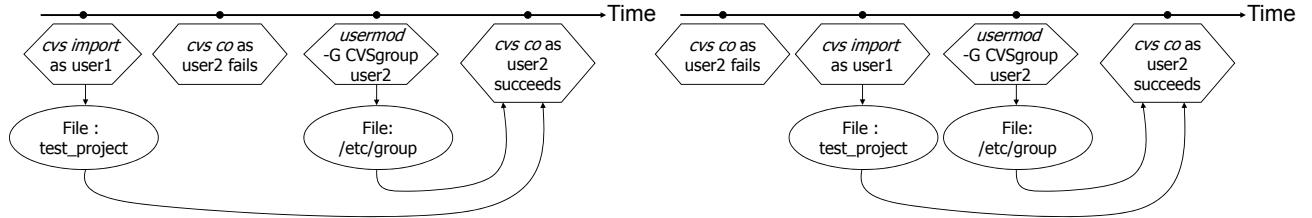
However, our heuristic is not foolproof. If a precondition is executed after failed predicates, it has the same causal effect as a solution. Thus, it is difficult to determine whether it is a precondition or a solution without the command semantics. Consider the same example, but with the order of "`cvs import as user1`" and "`cvs co as user2`" reversed, as shown in Figure 4. Both "`cvs import as user1`" and "`usermod -G CVSgroup user2`" have causal effects on the successful predicate and were executed after the failed predicate. Lacking semantic information about CVS, the precondition heuristic would mark both commands as part of the solution.

We believe that the second scenario is less likely to occur than the first because a user will typically set up the preconditions before executing a test case. In our user study, this heuristic worked in every case. However, we can also filter out incorrect solutions if they occur less frequently than correct ones, as described in the next section.

Our heuristic can work correctly even if a solution requires multiple commands. Consider a solution that requires two commands A and B, both of which occur chronologically after all failed predicates. Since both A and B have causal effects on the predicate, our heuristic would identify both as part of the solution.

5.4 Inferring solutions from user traces

Not all solutions identified by the precondition heuristic are the correct solution for the specific configuration problem the user is solving. Sometimes, a user does not



The `usermod -G CVSgroup user2` command adds user2 to the CVS group

Figure 3. Precondition heuristic example

fix the problem. Other times, a user fixes the problem incorrectly. To filter out such erroneous solutions, we rely on the observation made by PeerPressure [22] and other troubleshooting systems that the mass of users is typically right. Thus, solutions that occur more frequently are more likely to be correct.

We therefore rank solutions by the frequency that they occur in multiple traces. Effectively, solutions are ranked by their popularity; a solution is considered more popular if more users apply it to successfully fix their configuration problems. In our evaluation, this heuristic worked well because the most popular solutions found by users were the ones that solved the actual configuration problems that we introduced.

Counting the frequency of a solution’s popularity is complicated by the fact that different commands have the same effect. For example, when users in our study solved an Apache configuration problem caused by the execution bit on a user’s home directory (e.g., `/home/USERID`), being set incorrectly, their solutions included `chmod 755 /home/USERID`, `chmod 755 USERID/`, and `chmod og+rwx USERID`. Although these commands were syntactically different, they all had the same effect.

To better measure the popularity of a solution, we group solutions by their *state deltas*. The state delta captures the *difference* in the state of the system caused by the execution of a command by calculating the difference for each entity in the command’s output set. For example, the state delta for command `chmod a+r test.pl` includes only the change in file permissions for `test.pl`. The solution-ranking heuristic first groups solutions based on state deltas so that all solutions having the same state delta are in the same group. It then ranks the groups by their cardinality.

Here is an example of how the solution-ranking heuristic ranks three solutions: `chmod 755 /home/USERID`, `chmod 755 USERID/`, and `chmod -R 777 USERID/`. The the first two commands are placed into one group because they both have the same state delta (changing the permission of the directory

The `usermod -G CVSgroup user2` command adds user2 to the CVS group

Figure 4. Example of the precondition heuristic failing

CVS configuration problems	
1	Repository not properly initialized
2	User not added to CVS group
Apache Web server configuration problems	
1	Apache cannot search a user’s home directory due to incorrect permissions
2	Apache cannot read CGI scripts due to incorrect permissions

Table 2. Description of injected configuration problems

`/home/USERID to 755`). The third solution is put in a separate, less popular group.

6 Evaluation

To evaluate our methods for extracting predicates and solutions, we conducted a user study in which participants were asked to fix four configuration bugs: two for the CVS version control system and two for the Apache web server. These bugs are shown in Table 2. While these bugs may seem relatively simple, several users were unable to solve one or more of these bugs in the allotted time (15 minutes). We initially designed a user study with a few more complex bugs, but our initial feedback from trial users showed that the complex bugs were too hard to solve in a reasonable amount of time. We also decided to impose the 15-minute limit based on our trial study. Since we ask each user to solve a total of six problems (including two sample problems) and the study required some up-front explanation and paperwork, users needed to commit over two hours to participate. Even with the 15-minute limit, one user declined to complete all problems in the study.

6.1 Methodology

A total of twelve users with varying skills participated in our study: two experienced system administrators and ten graduate students, all from the University of Michigan. We asked participants to assess their experience level. Table 3 is a summary of our participants. For CVS, three participants (A,D, and K) rated themselves as experts, meaning that the participant had diagnosed

User	CVS version control			Apache Web server		
	Experience level	Prob 1 Fixed?	Prob 2 Fixed?	Experience level	Prob 1 Fixed?	Prob 2 Fixed?
A	Expert	N/A	Y	Expert	Y	Y
B	Novice	Y	Y	Intermediate	N	Y
C	Intermediate	Y	Y	Novice	Y	Y
D	Expert	Y	Y	Expert	Y	Y
E	Beginner	N	N	Expert	Y	N
F	Intermediate	Y	Y	Expert	Y	Y
G	Novice	Y	N/A	Beginner	N	N
H	Intermediate	Y	Y	Expert	Y	Y
I	Intermediate	Y	Y	Expert	Y	Y
J	Novice	Y	Y	Expert	Y	Y
K	Expert	Y	N	Intermediate	N	Y
L	Intermediate	Y	Y	Novice	N	Y
Total fixed		10	9		8	10

This table shows the experience of our participants and the number of problems solved.

Table 3. Participant summary

User	CVS problem 1			CVS problem 2		
	# of Pred	Correct?	Total # of cmdns	# of Pred	Correct?	Total # of cmdns
A	—	—	—	1	Yes	44
B	1	Yes	105	1	Yes	44
C	0	SBA	57	0	NRC	46
D	0	SBA	49	0	SBA	26
F	1	Yes	22	1	Yes	30
G	0	NRC	61	—	—	—
H	0	NRC	58	0	NRC	74
I	0	NRC	18	1	Yes	18
J	0	NRC	65	0	SBA	72
K	1	Yes	55	0	DNF	24
L	1	Yes	40	0	SBA	24

There are three reasons why no predicate was identified in some of the above traces: (1) NRC means that the user did not use a repeated command to test the configuration problem. (2) DNF means that the user did not fix the configuration problem. (3) SBA means that the user used a state-based approach.

Table 4. Summary of predicates generated for CVS

and fixed misconfigurations several times for that application. Five participants (C, F, H, I, and L) rated themselves as intermediates, meaning that the participant had fixed configurations for that application at least once, and three participants (B, G, and J) rated themselves as novices, meaning that the participant had used the application. Two participants (user E for CVS and user G for Apache) were listed as beginners because they were unfamiliar with the application. Both beginner participants were not asked to complete that portion of the study. For Apache, seven participants were experts (A, D, E, F, H, I, and J), two were intermediates (B and K), and two were novices (C and L). Additionally, user A did not complete CVS bug 1 because of an error we made in setting up the environment, and user G declined to do CVS problem 2.

User	Apache problem 1			Apache problem 2		
	# of Pred	Correct?	Total cmdns.	# of Pred	Correct?	Total cmdns.
A	1	Yes	45	1	Yes	45
B	0	DNF	39	0	NRC	39
C	2	Yes	41	1	Yes	41
		Yes				
D	1	Yes	68	1	Yes	68
E	0	NRC	35	1	No	35
F	0	NRC	33	1	Yes	33
H	1	Yes	32	1	Yes	32
I	1	Yes	22	1	Yes	22
J	0	NRC	40	1	Yes	40
K	1	No	67	1	Yes	67
L	0	DNF	55	0	NRC	55

There are two reasons why no predicate was identified in some of the above traces: (1) NRC means that the user did not use a repeated command to test the configuration problem. (2) DNF means that the user did not fix the configuration problem.

Table 5. Summary of predicates generated for Apache

For each application, each participant was first given a sample misconfiguration to fix. While fixing the sample, they could ask us questions. The participant then was asked to fix two additional misconfigurations without any guidance from us. We did not tell the participants any information about the bugs, so they needed to find the bug, identify the root cause, and fix it. Participants were given a maximum of 15 minutes to fix each problem. For each problem, 8–10 participants were able to fix the bug, but 1–3 were not.

For each task, our modified shell recorded traces of user input, system output, and the causality data (output sets) tracked by the kernel. The overhead of tracking the output set was shown to be negligible in our prior work [21]. We implemented the algorithms to analyze

User	Predicate	
B	Precond Pred Sol	export CVSROOT="/home/cvsroot" cvs import -m "Msg" yoyo/test_project yoyo start cvs -d /home/cvsroot init
F	Precond Pred Sol	export CVSROOT=/home/cvsroot cvs import test_project cvs -d /home/cvsroot init
K	Pred Sol	cvs -d /home/cvsroot import cvsroot cvs -d /home/cvsroot init
L	Pred Sol	cvs -d "/home/cvsroot" import test_project cvs -d /home/cvsroot init

Each predicate contains one or more steps. The last step of a predicate is listed as Pred, and the remaining steps are listed as Precond. We also list the solution (Sol) identified for each predicate.

Table 6. Correct predicates for CVS problem 1

these traces in Python. The computational cost is small; the time to analyze all traces averaged only 123 ms on a desktop computer with a 2.7 GHz processor and 1 GB memory. The shortest trace required 3 ms to analyze and the longest took 861 ms.

6.2 Quality of predicates generated

Table 4 and Table 5 summarize the predicates generated for each application. Each table lists the number of predicates extracted, whether the extracted predicates were correct, and the total number of commands entered by the user for that trace.

For both CVS bugs, we generated four correct predicates from ten traces, with no false positives. For Apache, we generated six correct predicates for the first bug and eight for the second, with one false positive for each bug. We are pleased that the results matched our design goal of minimizing false positives, while still extracting several useful predicates for each bug.

6.2.1 Predicates generated for CVS problem 1

The first CVS problem was caused by the CVS repository not being properly initialized, and the solution is to initialize it using `cvs init`. Table 6 lists the predicates we extracted. We break each predicate into two parts: the predicate (Pred), the repeated command that determines if this predicate returns true or false, and the precondition (Precond), the set of commands that need to be executed before the predicate. We also list the solution (Sol) identified by our methodology. All predicates that we identified involve the user importing a module into the CVS repository.

We did not identify predicates for participants C and D because they used a state-based approach in which they discovered that the CVS repository was not initialized by examining the repository directory. Participants G, H, I, and J used slightly different commands to test the system state before and after fixing the configuration problem, so our algorithm did not identify a predicate. For example, participant H's two `cvs import` commands had different CVS import comments.

6.2.2 Predicates generated for CVS problem 2

The second CVS bug was caused by a user not being in the CVS group, and the solution is to add that user to the group. Table 7 shows the predicates and solutions identified for this bug. All extracted predicates involve the user trying to check out a module from the CVS repository. These are multi-step predicates in which the checkout is proceeded by a CVS import command.

No predicate was generated for six traces. Participants C and H did not use repeated commands. For instance, participant C specified the root directory as `/home/cvsroot` in one instance and `/home/cvsroot/` in another. Participant H used the same command line but with different environment variables. Our algorithm was unable to identify a predicate for participant K because that user did not fix the problem. Finally, no predicate was found for participants D, J, and L because they used a state-based approach (participant D executed groups and participant L examined `/etc/group`).

6.2.3 Predicates generated for Apache problem 1

The first Apache bug was caused by Apache not having search permission for the user's home directory, and the solution is to change the directory permissions. Table 8 shows the correct predicates identified for each trace and the corresponding solutions. Almost all predicates download the user's home page. Some preconditions found by the precondition heuristic are not required for the predicate to work but also do not affect the correctness of the predicates. Predicate C-2, `apachectl stop` did not seem to be a correct predicate, so we examined why it was generated. We found that predicate C-2 successfully detected an error in the Apache configuration file introduced by participant C. `apachectl` is a script that controls the Apache process. It does not stop the Apache process if an error is detected in the configuration file. Thus, it is indeed a valid predicate.

No predicates were generated for some participants due to reasons similar to those seen in CVS traces. Participants B and L did not fix the problem. Participants E

User	Predicate	
A	Precond	cvs import test_project head start cvs co test_project export CVSROOT=/home/cvsroot cvs co test_project vi /etc/group
B	Precond Pred Sol	cvs -d /home/cvsroot import yoyo/test_project cvs -d /home/cvsroot checkout yoyo/test_project usermod -G cvsgroup USERID
F	Precond Pred Sol	cvs import test_project cvs co test_project export CVSROOT=/home/cvsroot cvs co test_project vim group
I	Precond Pred Sol	cvs -d /home/cvsroot import test_project cvs -d /home/cvsroot co test_project vi /etc/group

Each predicate contains one or more steps. The last step of a predicate is listed as Pred, and the remaining steps are listed as Precond. We also list the solution (Sol) identified for each predicate.

Table 7. Correct problems for CVS problem 2

User	Predicate		
A	Precond	wget http://localhost	
	Pred	chmod 777 index.html	
	Sol	chmod 777 public_html/ wget http://localhost/~USERID /index.html chmod 777 USERID	
C-1	Pred Sol	wget http://localhost/~USERID/ chmod o+r /home/USERID	
C-2	Pred Sol	apachectl stop vim /etc/httpd/conf/httpd.conf chmod o+r /home/USERID	
D	Precond Pred Sol	wget localhost chmod -R 777 public_html/ wget localhost/~USERID chmod -R 777 USERID/	
H	Precond	mkdir scratch	
	Pred Sol	wget http://localhost rm index.html wget http://localhost/~USERID chmod 755 /home/USERID/	
I	Precond Pred Sol	wget http://localhost wget http://localhost/~USERID chmod 755 /home/USERID/	

Each predicate contains one or more steps. The last step of a predicate is listed as Pred, and the remaining steps are listed as Precond. We also list the solution (Sol) identified for each predicate.

Table 8. Correct predicates for Apache problem 1

and F executed a command in different directories, so the base algorithm considered that command as not repeated.

One false positive was generated for participant K. Since participant K did not fix the problem, we first expected no predicate be generated. However, this participant edited the Apache configuration file multiple times using emacs. The file contains an error message that was displayed only in some executions of emacs in which the participant scrolled down. Additionally, the participant sometimes modified the file and other times did not, leading to different output sets. Since two output features

User	Predicate	
A	Pred Sol	wget http://localhost/cgi-bin/test.pl chmod 755 test.pl
C	Pred Sol	wget http://localhost/cgi-bin/test.pl chmod go+r test.pl
D	Precond	wget localhost/cgi-bin/test.pl
	Pred Sol	vi /var/www/cgi-bin/test.pl wget localhost/cgi-bin/test.pl vi /var/www/cgi-bin/test.pl chmod 777 /var/www/cgi-bin/test.pl
F	Pred Sol	wget http://localhost/cgi-bin/test.pl chmod 755 test.pl
H	Precond Pred Sol	mkdir scratch wget http://localhost/cgi-bin/test.pl chmod 755 /var/www/cgi-bin/test.pl
I	Pred Sol	wget http://localhost/cgi-bin/test.pl chmod 755 /var/www/cgi-bin/test.pl
J	Pred Sol	wget 127.0.0.1/cgi-bin/test.pl chmod +r test.pl
K	Pred Sol	wget http://localhost/cgi-bin/test.pl chmod a+r test.pl

Each predicate contains one or more steps. The last step of a predicate is listed as Pred, and the remaining steps are listed as Precond. We also list the solution (Sol) identified for each predicate.

Table 9. Correct predicates for Apache problem 2

changed, our heuristic labeled this command a predicate.

6.2.4 Predicates generated for Apache problem 2

The second Apache problem was caused by Apache not having read permission for a CGI Perl script and the solution is to change the file permission. The initial state included a default CGI Perl script in Apache's default CGI directory, /var/www/cgi-bin. Table 9 shows the correctly identified predicates, which all involve retrieving the the CGI script. Some traces include preconditions that are not really required, but these preconditions do not affect the correctness of the predicates.

No predicate was generated for participants B and L because they executed wget in different directories. The

	Solution	Freq.
1	cvs -d /home/cvsroot init as cvsroot	3
2	vi /etc/group as root	3
3	cvs -d /home/cvsroot init as root	1
4	usermod -G cvsgroup USERID as root	1

Table 10. Solutions ranked for CVS

	Solution	Freq.
1	chmod 755 /var/www/cgi-bin/test.pl	6
2	chmod 755 /home/USERID as root	2
3	chmod 777 USERID as root	1
4	chmod o+rwx /home/USERID as root	1
5	chmod -R 777 USERID/ as USERID	1
6	vim /etc/httpd/conf/httpd.conf as root	1
7	chmod 777 /var/www/cgi-bin/test.pl	1
8	chmod +r test.pl as root	1
9	vi /var/www/cgi-bin/test.pl	1

Table 11. Solutions ranked for Apache

predicate identified for participant E was incorrect. Participant E did not fix the problem. However, participant E used `links` to connect to the Apache Web server. This utility generates files with random names so each invocation has a different output set. Combined with an error message being generated in some instances but not others, this led our heuristic to falsely identify a predicate. Based on our two false positives, we believe it would be beneficial to ask users if the problem was fixed at the end of troubleshooting and not try to generate predicates if it was not. This question may not be too intrusive and would eliminate both false positives.

6.3 Solution ranking results

We took the solutions found by the precondition heuristic from all traces and used our solution-ranking heuristic to rank them by frequency. Table 10 shows results for CVS. The two highest ranked solutions are the correct fixes for the two problems we introduced. Note that the solutions are captured as a state delta, so a command that starts an editor (`vi`) is really a patch to the edited file. The third solution listed is less correct than the first because it creates the repository as root, giving incorrect permissions. The final solution is as correct as the second for fixing CVS problem 2.

Table 11 shows results for Apache. The two highest ranked solutions fix Apache problems 2 and 1, respectively. Solution 3-5 are less correct than solution 1 because they grant more than the minimum permission required. The 6th solution correctly solves the bug introduced by participant C for Apache problem 1. The remaining 3 solutions are less correct than solution 1 because they either give too much permission or do not fix the problem.

7 Discussion

We first discuss the challenges we encountered and how we would like to address them, followed by discussing the limitations of our approach.

7.1 Challenges

One challenge we encountered relates to canonicalization. Our troubleshooting shell canonicalizes common environment variables, such as home directories and user names. However, applications may also use temporary files, specific file location settings, or other environment variables. More thought may be required on how to handle application-specific variables if application semantic knowledge is not presented.

Our current study simplifies the configuration problem by restricting user input to text form (i.e., by requiring the activity to occur within the scope of a Unix shell). We chose this approach to speed the implementation of our prototype. Writing tools to capture text input and output is easier than writing tools to capture graphical interactions.

We sketch here how we would extend our current approach to handle GUI applications. First, a command roughly maps to an action in the command line interface. However, users launch a GUI application to explore many different configuration actions more flexibly, which makes it hard to find repeated tasks for our base algorithm. Without user input, it may be hard to break a long GUI session into individual actions that are more likely to repeat. Second, if the user happens to execute one action at a time using a GUI application, we need a more sophisticated way to identify if two GUI sessions are the same. One possible solution is to use state deltas to capture the effect of performing GUI applications and compare such deltas. We would capture output features as follows:

- **Exit value.** Since GUI applications are designed to execute several actions, they usually do not return the proper value to the calling shell.
- **Screen output.** GUI configuration tools may offer additional semantic information. For instance, error dialogs are a common widget that indicate the failure of an operation. Such dialogs can be queried using the GUI's accessibility APIs (intended to benefit vision-impaired users).
- **Output set.** We could capture the output set for a GUI application in the same way as a command.

7.2 Limitations

Our troubleshooting shell assumes that all configuration actions happen under its purview. Configuration problems involving external components, such as printer

or network communication, are not handled by our shell because it does not have the ability to track external components' output. Also, one can imagine a precondition command executed before our troubleshooting shell is launched; our shell will not find that precondition as it limits dependency tracking to the period in which it is running.

8 Conclusion

Predicates play an important role for automated configuration management tools such as AutoBash and Chronus. However, writing predicates by hand is tedious, time-consuming, and requires expert knowledge. This work solves the problem of manually writing predicates by automatically inferring predicates and solutions from traces of users fixing configuration problems.

Acknowledgments

We thank our shepherd, Yinglian Xie, and the anonymous reviewers for valuable feedback on this paper, as well as our user study participants. The work has been supported by the National Science Foundation under award CNS-0509093. Jason Flinn is supported by NSF CAREER award CNS-0346686. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

References

- [1] ATTARIYAN, M., AND FLINN, J. Using causality to diagnose configuration bugs. In *Proceedings of the USENIX Annual Technical Conference* (Boston, MA, June 2008), pp. 171–177.
- [2] BEREITER, S., AND MILLER, S. *Troubleshooting and Human Factors in Automated Manufacturing Systems*. Noyes Publications, March 1989.
- [3] BOYAPATI, C., KHURSHID, S., AND MARINOV, D. Korat: Automated testing based on Java predicates. In *Proceedings of ACM International Symposium on Software Testing and Analysis ISSTA 2002* (2002).
- [4] CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 4, 3 (1978), 178–187.
- [5] COMPUTING RESEARCH ASSOCIATION. Final report of the CRA conference on grand research challenges in information systems. Tech. rep., September 2003.
- [6] DALAL, S. R., JAIN, A., KARUNANITHI, N., LEATON, J. M., LOTT, C. M., PATTON, G. C., AND HOROWITZ, B. M. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)* (Los Angeles, California, May 1999).
- [7] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001), pp. 57–72.
- [8] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)* (2005), pp. 213–223.
- [9] HA, J., ROSSBACH, C. J., DAVIS, J. V., ROY, I., RAMADAN, H. E., PORTER, D. E., CHEN, D. L., AND WITCHEL, E. Improved error reporting for software that uses black-box components. In *Proceedings of the Conference on Programming Language Design and Implementation 2007* (San Diego, CA, 2007).
- [10] HAREL, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 3 (1987), 231–274.
- [11] JULIA, H., TRALAMAZZA, D., ZAMFIR, C., AND CANDEA, G. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (San Diego, CA, December 2008).
- [12] KAPOOR, A. Web-to-host: Reducing total cost of ownership. Tech. Rep. 200503, The Tolly Group, May 2000.
- [13] MARINOV, D., AND KHURSHID, S. Testera: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)* (San Diego, CA, November 2001).
- [14] MEMON, A. M., POLLACK, M. E., AND SOFFA, M. L. Automated test oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, 2000), pp. 30–39.
- [15] MICKENS, J., SZUMMER, M., AND NARAYANAN, D. Snitch: Interactive decision trees for troubleshooting misconfigurations. In *In Proceedings of the 2007 Workshop on Tackling Computer Systems Problems with Machine Learning Techniques* (Cambridge, MA, April 2007).
- [16] NAGARAJA, K., OLIVERIA, F., BIANCHINI, R., MARTIN, R., AND NGUYEN, T. Understanding and dealing with operator mistakes in Internet services. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 61–76.
- [17] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.
- [18] OFFUTT, J., AND ABDURAZIK, A. Generating tests from uml specifications. In *Second International Conference on the Unified Modeling Language (UML99)* (Ocrober 1999).
- [19] RICHARDSON, D. J. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)* (Seattle, WA, 94), pp. 138–153.
- [20] RICHARDSON, D. J., AHA, S. L., AND O'MALLEY, T. O. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering* (Melbourne, Australia, 1992), pp. 105–118.
- [21] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, October 2007), pp. 237–250.
- [22] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 245–257.
- [23] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the USENIX Large Installation Systems Administration Conference* (October 2003), pp. 159–172.
- [24] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 77–90.

JustRunIt: Experiment-Based Management of Virtualized Data Centers

Wei Zheng,[†] Ricardo Bianchini,[†] G. John Janakiraman,[‡] Jose Renato Santos,[‡] and Yoshio Turner[‡]

[†]*Department of Computer Science
Rutgers University
{wzheng,ricardob}@cs.rutgers.edu}*

[‡]*HP Labs
Hewlett-Packard Corporation
{joserenato.santos,yoshio.turner}@hp.com*

Abstract

Managing data centers is a challenging endeavor. State-of-the-art management systems often rely on analytical modeling to assess the performance, availability, and/or energy implications of potential management decisions or system configurations. In this paper, we argue that actual experiments are cheaper, simpler, and more accurate than models for many management tasks. To support this claim, we built an infrastructure for experiment-based management of virtualized data centers, called JustRunIt. The infrastructure creates a sandboxed environment in which experiments can be run—on a very small number of machines—using real workloads and real system state, but without affecting the on-line system. Automated management systems or the system administrator herself can leverage our infrastructure to perform management tasks on the on-line system. To evaluate the infrastructure, we apply it to two common tasks: server consolidation/expansion and evaluating hardware upgrades. Our evaluation demonstrates that JustRunIt can produce results realistically and transparently, and be nicely combined with automated management systems.

1 Introduction

Managing data centers is a challenging endeavor, especially when done manually by system administrators. One of the main challenges is that performing many management tasks involves selecting a proper resource allocation or system configuration out of a potentially large number of possible alternatives. Even worse, evaluating each possible management decision often requires understanding its performance, availability, and energy consumption implications. For example, a common management task is to partition the system’s resources across applications to optimize performance and/or energy consumption, as is done in server consolidation and virtual machine (VM) placement. Another example is the evaluation of software or hardware upgrades, which involves

determining whether application or system behavior will benefit from the candidate upgrades and by how much. Along the same lines, capacity planning is a common management task that involves selecting a proper system configuration for a set of applications.

Previous efforts have automated resource-partitioning tasks using simple heuristics and/or feedback control, e.g. [1, 6, 18, 21, 28, 29]. These policies repeatedly adjust the resource allocation to a change in system behavior, until their performance and/or energy goals are again met. Unfortunately, when this react-and-observe approach is not possible, e.g. when evaluating software or hardware upgrades, these policies cannot be applied.

In contrast, analytical modeling can be used to automate all of these management tasks. Specifically, modeling can be used to predict the impact of the possible management decisions or system configurations on performance, availability, and/or energy consumption. With these predictions, the management system can make the best decision. For example, researchers have built resource-partitioning systems for hosting centers that use models to predict throughput and response time, e.g. [9, 27]. In addition, researchers have built systems that use models to maximize energy conservation in data centers, e.g. [7, 13]. Finally, researchers have been building models that can predict the performance of Internet applications on CPUs with different characteristics [23]; such models can be used in deciding whether to upgrade the server hardware.

Performance models are often based on queuing theory, whereas availability models are often based on Markovian formalisms. Energy models are typically based on simple (but potentially inaccurate) models of power consumption, as a function of CPU utilization or CPU voltage/frequency. On the bright side, these models are useful in data center management as they provide insight into the systems’ behaviors, can be solved quickly, and allow for large parameter space explorations. Essentially, the models provide an efficient way of answering

“what-if” questions during management tasks.

Unfortunately, modeling has a few serious shortcomings. First, modeling consumes a very expensive resource: highly skilled human labor to produce, calibrate, and validate the models. Second, the models typically rely on simplifying assumptions. For example, memoryless arrivals is a common assumption of queuing models for Internet services [24]. However, this assumption is invalid when requests come mostly from existing sessions with the service. Another common simplifying assumption is the cubic relationship between CPU frequency and power consumption [7]. With advances in CPU power management, such as clock gating, the exact power behavior of the CPU is becoming more complex and, thus, more difficult to model accurately. Third, the models need to be re-calibrated and re-validated as the systems evolve. For example, the addition of new machines to a service requires queuing models to be calibrated and validated for them.

Given these limitations, in this paper we argue that actual experiments are a better approach than modeling for supporting many management tasks. Actual experiments exchange an expensive resource (human labor) for much cheaper ones (the time and energy consumed by a few machines in running the experiments). Moreover, they do not rely on simplifying assumptions or require calibration and validation. Thus, actual experiments are cheaper, simpler, and more accurate than models in their ability to answer “what-if” questions. We further argue that the experiments can be performed in a flexible, realistic, and transparent manner by leveraging current virtualization technology.

To support our claims in a challenging environment, we built JustRunIt, an infrastructure for experiment-based management of virtualized data centers hosting multiple Internet services. JustRunIt creates a sandboxed environment in which experiments can be run on a small number of machines (e.g., one machine per tier of a service) without affecting the on-line system. JustRunIt clones a small subset of the on-line VMs (e.g., one VM per tier of the service) and migrates them to the sandbox. In the sandbox, JustRunIt precisely controls the resources allocated to the VMs, while offering the same workload to them that is offered to similar VMs on-line. Workload duplication is implemented by JustRunIt’s server proxies. For flexibility, the administrator can specify the resources (and the range of allocations) with which to experiment and how long experiments should be run. If there is not enough time to run all possible experiments (i.e., all combinations of acceptable resource allocations), JustRunIt uses interpolation between actual experimental results to produce the missing results but flags them as potentially inaccurate.

Automated management systems or the system admin-

istrator can use the JustRunIt results to perform management tasks on the on-line system. If any interpolated results are actually used by the system or administrator, JustRunIt runs the corresponding experiments in the background and warns the administrator if any experimental result differs from the corresponding interpolated result by more than a threshold amount.

To evaluate our infrastructure, we apply it to systems that automate two common management tasks: server consolidation/expansion and evaluation of hardware upgrades. Modeling has been used in support of both tasks [7, 24], whereas feedback control is only applicable for some cases of the former [7]. JustRunIt combines nicely with both systems. Our evaluation demonstrates that JustRunIt can produce results realistically and transparently, enabling automated management systems to perform their tasks effectively. In fact, JustRunIt can produce system configurations that are as good as those resulting from idealized, perfectly accurate models, at the cost of the time and energy dedicated to experiments.

The remainder of the paper is organized as follows. The next section describes JustRunIt in detail. Section 3 describes the automated management systems that we designed for our two case studies. Section 4 presents our evaluation of JustRunIt and the results of our case studies. Section 5 overviews the related work. Finally, Section 6 draws our conclusions, discusses the limitations of JustRunIt, and mentions our future work.

2 JustRunIt Design and Implementation

2.1 Target Environment

Our target environment is virtualized data centers that host multiple independent Internet services. Each service comprises multiple tiers. For instance, a typical three-tier Internet service has a Web tier, an application tier, and a database tier. Each tier may be implemented by multiple instances of a software server, e.g. multiple instances of Apache may implement the first tier of a service. Each service has strict response-time requirements specified in SLAs (Service Level Agreements) negotiated between the service provider and the data center.

In these data centers, all services are hosted in VMs for performance and fault isolation, easy migration, and resource management flexibility. Moreover, each software server of a service is run on a different VM. VMs hosting software servers from different services may co-locate on a physical machine (PM). However, VMs hosting software servers from the same service tier are hosted on different PMs for high availability. All VMs have network-attached storage provided by a storage server.

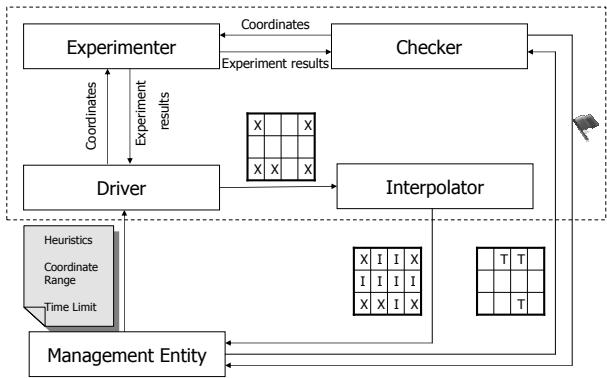


Figure 1: Overview of JustRunIt. “X” represents a result obtained through experimentation, whereas “I” represents an interpolated result. “T” represents an interpolated result that has been used by the management entity.

2.2 System Infrastructure

Figure 1 shows an overview of the system infrastructure of JustRunIt. There are four components: experimenter, driver, interpolator, and checker. The *experimenter* implements the VM cloning and workload duplication mechanism to run experiments. Each experiment tests a possible configuration change to a cloned software server under the current live workload. A configuration change may be a different resource allocation (e.g., a larger share of the CPU) or a different hardware setting (e.g., a higher CPU voltage/frequency). The results of each experiment are reported as the server throughput, response time, and energy consumption observed under the tested configuration.

The experiment *driver* chooses which experiments to run in order to efficiently explore the configuration parameter space. The driver tries to minimize the number of experiments that must be run while ensuring that all the experiments complete within a user-specified time bound. The driver and experimenter work together to produce a matrix of experimental results in the configuration parameter space. The coordinates of the matrix are the configuration parameter values for each type of resource, and the values recorded at each point are the performance and energy metrics observed for the corresponding resource assignments.

Blank entries in the matrix are filled in by the *interpolator*, based on linear interpolation from the experimental results in the matrix. The filled matrix is provided to the management entity—i.e., the system administrator or an automated management system—for use in deciding resource allocations for the production system.

If the management entity uses any of the interpolated performance or energy values, the *checker* invokes the

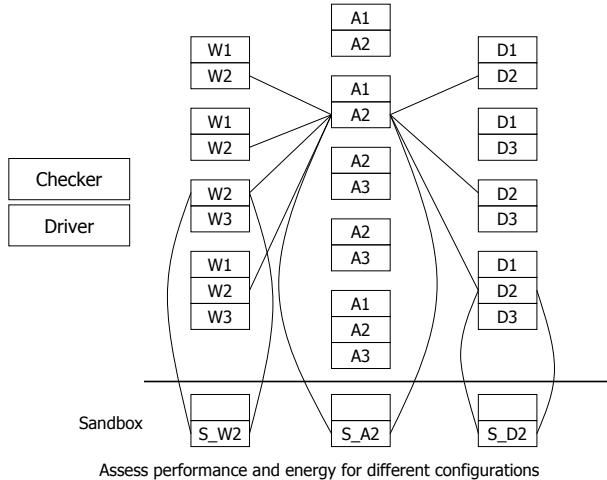


Figure 2: Virtualized data center and JustRunIt sandbox. Each box represents a VM, whereas each group of boxes represents a PM. “W2”, “A2”, and “D2” mean Web, application, and database server of service 2, respectively. “S_A2” means sandboxed application server of service 2.

experimenter to run experiments to validate those values. If it turns out that the difference between the experimental results and the interpolated results exceeds a user-specified threshold value, then the checker notifies the management entity.

We describe the design of each component of JustRunIt in detail in the following subsections.

2.2.1 Experimenter

To run experiments, the experimenter component of JustRunIt transparently clones a subset of the live production system into a sandbox and replays the live workload to the sandbox system. VM cloning instantly brings the sandbox to the same operational state as the production system, complete with fully warmed-up application-level and OS-level caches (e.g., file buffer cache). Thus, tests can proceed with low startup time on a faithful replica of the production system. By cloning only a subset of the system, JustRunIt minimizes the physical resources that must be dedicated to testing. Workload replay to the sandbox is used to emulate the timing and functional behavior of the non-duplicated portions of the system.

The use of JustRunIt in a typical virtualized data center is illustrated in Figure 2. The figure shows VMs of multiple three-tier services sharing each PM. Each service tier has multiple identically configured VMs placed on different PMs. (Note that VMs of one tier do not share PMs with VMs of other tiers in the figure. Although JustRunIt is agnostic to VM placement, this restriction on VM placement is often used in practice to reduce software licensing costs [18].) For simpler management, the

set of PMs in each tier is often homogeneous.

The figure also shows one VM instance from each tier of service 2 being cloned into the sandbox for testing. This is just an example use of JustRunIt; we can use different numbers of PMs in the sandbox, as we discuss later. Configuration changes are applied to the clone server, and the effects of the changes are tested by replaying live traffic duplicated from the production system. The sandbox system is monitored to determine the resulting throughput, response time, and energy consumption. The experimenter reports these results to the driver to include in the matrix described in Section 2.2. If experiments are run with multiple service tiers, a different matrix will be created for each tier.

Although it may not be immediately obvious, the experimenter assumes that the virtual machine monitor (VMM) can provide performance isolation across VMs and includes non-work-conserving resource schedulers. These features are required because the experiments performed in the sandbox must be realistic representations of what would happen to the tested VM in the production system, regardless of any other VMs that may be co-located with it. We can see this by going back to Figure 2. For example, the clone VM from the application tier of service 2 must behave the same in the sandbox (where it is run alone on a PM) as it would in the production system (where it is run with A1, A3, or both), given the same configuration. Our current implementation relies on the latest version of the Xen VMM (3.3), which provides isolation for the setups that we consider.

Importantly, both performance isolation and non-work-conserving schedulers are desirable characteristics in virtualized data centers. Isolation simplifies the VM placement decisions involved in managing SLAs, whereas non-work-conserving schedulers allow more precise resource accounting and provide better isolation [18]. Most critically, both characteristics promote performance predictability, which is usually more important than achieving the best possible performance (and exceeding the SLA requirements) in hosting centers.

Cloning. Cloning is accomplished by minimally extending standard VM live migration technology [8, 16]. The Xen live migration mechanism copies dirty memory pages of a running VM in the background until the number of dirty pages is reduced below a predefined threshold. Then VM execution is paused for a short time (tens of milliseconds) to copy the remaining dirty pages to the destination. Finally, execution transfers to the new VM, and the original VM is destroyed. Our cloning mechanism changes live migration to resume execution on both the new VM and the original VM.

Since cloning is transparent to the VM, the clone VM inherits the same network identity (e.g., IP/MAC addresses) as the production VM. To avoid network address

conflicts, the cloning mechanism sets up network address translation to transparently give the clone VM a unique external identity exposed to the network while concealing the clone VM’s internal addresses. We implemented this by extending Xen’s backend network device driver (“netback”) to perform appropriate address translations and protocol checksum corrections for all network traffic to and from the clone VM.

The disk storage used by the clone VMs must also be replicated. During the short pause of the production system VM at the end of state transfer, the cloning mechanism creates a copy-on-write snapshot of the block storage volumes used by the production VM, and assigns them to the clone VM. We implemented this using the Linux LVM snapshot capability and by exporting volumes to VMs over the network using iSCSI or ATA Over Ethernet. Snapshotting and exporting the storage volumes incurs only a sub-second delay during cloning. Storage cloning is transparent to the VMs, which see logical block devices and do not know that they are accessing network storage.

JustRunIt may also be configured *not* to perform VM cloning in the sandbox. This configuration allows it to evaluate upgrades of the server software (e.g., Apache), operating system, and/or service application (as long as the application upgrade does not change the application’s messaging behavior). In these cases, the management entity has to request experiments that are long enough to amortize any cold-start caching effects in the sandbox execution. However, long experiments are not a problem, since software upgrades typically do not have stringent time requirements.

Proxies. To carry out testing, the experimenter replays live workload to the VMs in the sandbox. Two low-overhead proxies, called in-proxy and out-proxy, are inserted into communication paths in the production system to replicate traffic to the sandbox. The proxies are application protocol-aware and can be (almost entirely) re-used across services that utilize the same protocols, as we detail below. The in-proxy mimics the behavior of all the previous tiers before the sandbox, and the out-proxy mimics the behavior of all the following tiers. The local view of a VM, its cloned sandbox VM, and the proxies is shown in Figure 3.

After cloning, the proxies create as many connections with the cloned VM as they have with the original VM. The connections that were open between the proxies and the original VM at the time it was cloned will timeout at the cloned VM. In fact, no requests that were active in the original VM at the time of cloning get successfully processed at the cloned VM.

The in-proxy intercepts requests from previous tiers to the tested VM. When a request arrives, the in-proxy records the request (*Reqn* in Figure 3) and its arrival

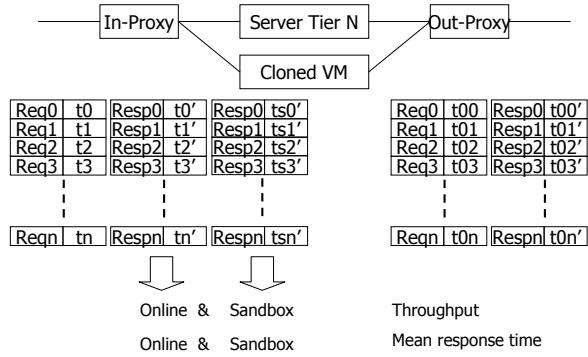


Figure 3: Cloned VM and proxy data structures.

time (tn). The in-proxy forwards the request to the on-line production system and also sends a duplicate request to the sandbox for processing. To prevent the sandbox system from running ahead of the production system, the transmission of the duplicate request is delayed by a fixed time interval (it is sufficient for the fixed time shift to be set to any value larger than the maximum response time of the service plus the cloning overhead). Both systems process the duplicated requests and eventually generate replies that are intercepted by the in-proxy. For the reply from the production system, the in-proxy records its arrival time (Tn') and forwards the reply back to the previous tier. Later, when the corresponding reply from the sandbox arrives, the in-proxy records its arrival time (tsn'). The arrival times are used to measure the response times of the production and sandbox systems.

The production and sandbox VMs may need to send requests to the next tier to satisfy a request from the previous tier. These (duplicated) requests are intercepted by the out-proxy. The out-proxy records the arrival times ($t0n$) and content of the requests from the production system, and forwards them to the next tier. The out-proxy also records the arrival times ($t0n'$) and content of the corresponding replies, and forwards them to the production system. When the out-proxy receives a request from the sandbox system, it uses hash table lookup to find the matching request that was previously received from the production system. (Recall that the matching request will certainly have been received because the replay to the sandbox is time-shifted by more than the maximum response time of the service.) The out-proxy transmits the recorded reply to the sandbox after a delay. The delay is introduced to accurately mimic the delays of the subsequent tiers and is equal to the delay that was previously experienced by the production system ($t0n' - t0n$) for the same request-reply pair.

At the end of an experiment, the in-proxy reports the throughput and response time results for the production and sandbox systems. The throughput for each system is determined by the number of requests successfully

served from the tiers following the in-proxy. The response time for each system is defined as the delay after a request arrives to the in-proxy until its reply is received. Since out-proxies enforce that the delays of subsequent tiers are equal for the production and sandbox system, the difference of throughput and response time between the production and sandbox systems is the performance difference between the original VM and cloned VM.

The proxies can be installed dynamically anywhere in the system, depending on which VMs the management entity may want to study at the time. However, we have only implemented in-proxies and out-proxies for Web and application servers so far. Cross-tier interactions between proxies, i.e. the communication between the out-proxy of the Web tier and the in-proxy of the application tier, occur in exactly the same way as the communication between regular servers.

In future work, we plan to implement an in-proxy for database servers by borrowing code from the Clustered-JDBC (C-JDBC) database middleware [5]. Briefly, C-JDBC implements a software controller between a JDBC application and a set of DBMSs. In its full-replication mode, C-JDBC keeps the content of the database replicated and consistent across the DBMSs. During experimentation, our in-proxy will do the same for the on-line and sandboxed DBMSs. Fortunately, C-JDBC already implements the key functionality needed for cloning, namely the ability to integrate the sandboxed DBMS and update its content for experimentation. To complete our in-proxy, we plan to modify C-JDBC to record the on-line requests and later replay them to the sandboxed DBMS. We have modified C-JDBC in similar ways [17].

Non-determinism. A key challenge for workload replay is to tolerate non-deterministic behavior in the production and sandbox systems. We address non-determinism in three ways. First, to tolerate network layer non-determinism (e.g., packet drops) the proxies replicate application-layer requests and replies instead of replicating network packets directly.

Second, the replay is implemented so that the sandboxed servers can *process* requests and replies in a different order than their corresponding on-line servers; only the timing of the message *arrivals* at the sandboxed servers is guaranteed to reflect that of the on-line servers. This ordering flexibility tolerates non-determinism in the behavior of the software servers, e.g. due to multithreading. However, note that this flexibility is only acceptable for Web and application-tier proxies, since requests from different sessions are independent of each other in those tiers. We will need to enforce ordering more strictly in the in-proxy for database servers, to prevent the original and cloned databases from diverging. Our in-proxy will do so by forcing each write (and commit) to execute by itself *during experimentation only* forcing a complete

ordering between all pairs of read-write and write-write operations; concurrent reads will be allowed to execute in any order. We have successfully created this strict ordering in C-JDBC before [17] and saw no noticeable performance degradation for one of the services we study in this paper.

Third, we tolerate application-layer non-determinism by designing the proxies to be application protocol-aware (e.g., the Web server in-proxies understand HTTP message formats). The proxies embody knowledge of the fields in requests and replies that can have non-deterministic values (e.g., timestamps, session IDs). When the out-proxy sees a non-deterministic value in a message from the sandbox, the message is matched against recorded messages from the production system using wildcards for the non-deterministic fields.

Our study of two services (an auction and a bookstore) shows that our proxies effectively tolerate their non-determinism. Even though some messages in these services have identical values except for a non-deterministic field, our wildcard mechanism allows JustRunIt to properly match replies in the production and sandbox systems for two reasons. First, all replies from the sandbox are dropped by the proxies, preventing them from disrupting the on-line system. Second, using different replies due to wildcard mismatch does not affect the JustRunIt results because the replies are equivalent and all delays are still accounted for.

We plan to study non-determinism in an even broader range of services. In fact, despite our promising experience with the auction and bookstore services, some types of non-determinism may be hard for our proxies to handle. In particular, services that non-deterministically change their messaging behavior (not just particular fields or the destination of the messages) or their load processing behavior (e.g., via non-deterministic load-shedding) would be impossible to handle. For example, a service in which servers may send an unpredictable number of messages in response to each request cannot be handled by our proxies. We have not come across any such services, though.

2.2.2 Experiment Driver

Running experiments is not free. They cost time and energy. For this reason, JustRunIt allows the management entity to configure the experimentation using a simple configuration file. The entity can specify the tier(s) with which JustRunIt should experiment, which experiment heuristics to apply (discussed below), which resources to vary, the range of resource allocations to consider, how many equally separated allocation points to consider in the range, how long each experiment should take, and how many experiments to run. These parameters can di-

rectly limit the time and indirectly limit the energy consumed by the experiments, when there are constraints on these resources (as in Section 3.1). When experiment time and energy are not relevant constraints (as in Section 3.2), the settings for the parameters can be looser.

Based on the configuration information, the experiment driver directs the experimenter to explore the parameter space within the time limit. The driver starts by running experiments to fill in the entries at the corners of the result matrix. For example, if the experiments should vary the CPU allocation and the CPU frequency, the matrix will have two dimensions and four corners: (min CPU alloc, min CPU freq), (min CPU alloc, max CPU freq), (max CPU alloc, min CPU freq), and (max CPU alloc, max CPU freq). The management entity must configure JustRunIt so at least these corner experiments can be performed. After filling in the corner coordinates, the driver then proceeds to request experiments exactly in the middle of the unexplored ranges defined by each resource dimension. After those are performed, it recursively subdivides the unexplored ranges in turn. This process is repeated until the number of experiments requested by the management entity have been performed or there are no more experiments to perform.

We designed two heuristics for the driver to use to avoid running unnecessary experiments along each matrix dimension. The two observations behind the heuristics are that: 1) beyond some point, resource additions do not improve performance; 2) the performance gain for the same resource addition to different tiers will not be the same, and the gains drop consistently and continually (diminishing returns).

Based on observation 1), the first heuristic cancels the remaining experiments with larger resource allocations along the current resource dimension, if the performance gain from a resource addition is less than a threshold amount. Based on observation 2), the second heuristic cancels the experiments with tiers that do not produce the largest gains from a resource addition. As we add more resources to the current tier, the performance gains decrease until some other tier becomes the tier with the largest gain from the same resource addition. For example, increasing the CPU allocation on the bottleneck tier, say the application tier, will significantly improve overall response time. At some point, however, the bottleneck will shift to other tiers, say the Web tier, at which point the driver will experiment with the Web tier and gain more overall response time improvement with the same CPU addition.

2.2.3 Interpolator and Checker

The interpolator predicts performance results for points in the matrix that have not yet been determined through

experiments. For simplicity, we use linear interpolation to fill in these blanks, and we mark the values to indicate that they are just interpolated.

If the management entity uses any interpolated results, the checker tries to verify the interpolated results by invoking the experimenter to run the corresponding experiments in the background. If one of these background experimental results differs from the corresponding interpolated result by more than a user-specified threshold value, the checker raises a flag to the management entity to decide how to handle this mismatch.

The management entity can use this information in multiple ways. For example, it may reconfigure the driver to run more experiments with the corresponding resources from now on. Another option would be to reconfigure the range of allocations to consider in the experiments from now on.

2.3 Discussion

Uses of JustRunIt. We expect that JustRunIt will be useful for many system management scenarios. For example, in this paper we consider resource management and hardware upgrade case studies. In these and other scenarios, JustRunIt can be used by the management entity to safely, efficiently, and realistically answer the same “what-if” questions that modeling can answer given the current workload and load intensity.

Moreover, like modeling, JustRunIt can benefit from load intensity prediction techniques to answer questions about future scenarios. JustRunIt can do so because its request replay is shifted in time and can be done at any desired speed. (Request stream acceleration needs to consider whether requests belong to an existing session or start a new session. JustRunIt can properly accelerate requests because it stores enough information about them to differentiate between the two cases.) Section 6 discusses how the current version of JustRunIt can be modified to answer “what-if” questions about different workload mixes as well.

Although our current implementation does not implement this functionality, JustRunIt could also be used to select the best values for software tunables, e.g. the number of threads or the size of the memory cache in Web servers. Modeling does not lend itself directly to this type of management task. Another possible extension could be enabling JustRunIt to evaluate the correctness of administrator actions, as in action-validation systems [15, 17]. All the key infrastructure required by these systems (i.e., proxies, cloning, sandboxing) is already part of the current version of JustRunIt, so adding the ability to validate administrator actions should be a simple exercise. Interestingly, this type of functionality cannot be provided by analytic models or feedback control.

Obviously, JustRunIt can answer questions and validate administrator actions at the cost of experiment time and energy. However, note that the physical resources required by JustRunIt (i.e., enough computational resources for the proxies and for the sandbox) can be a very small fraction of the data center’s resources. For example, in Figure 2, we show that just three PMs are enough to experiment with all tiers of a service at the same time, regardless of how large the production system is. Even fewer resources, e.g. one PM, can be used, as long as we have the time to experiment with VMs sequentially. Furthermore, the JustRunIt physical resources can be borrowed from the production system itself, e.g. during periods of low load.

In essence, JustRunIt poses an interesting tradeoff between the amount of physical resources it uses, the experiment time that needs to elapse before decisions can be made, and the energy consumed by its resources. More physical resources translate into shorter experiment times but higher energy consumption. For this reason, we allow the management entity to configure JustRunIt in whatever way is appropriate for the data center.

Engineering cost of JustRunIt. Building the JustRunIt proxies is the most time-consuming part of its implementation. The proxies must be designed to properly handle the communication protocols used by services. Our current proxies understand the HTTP, mod_jk, and MySQL protocols. We have built our proxies starting from the publicly available Tinyproxy HTTP proxy daemon [2]. Each proxy required only between 800 and 1500 new lines of C code. (VM cloning required 42 new lines of Python code in the xend control daemon and the xm management tool, whereas address translation required 244 new lines of C code in the netback driver.) The vast majority of the difference between Web and application server proxies comes from their different communication protocols.

The engineering effort required by the proxies can be amortized, as they can be reused for any service based on the same protocols. However, the proxies may need modifications to handle any non-determinism in the services themselves. Fortunately, our experience with the auction and bookstore services suggests that the effort involved in handling service-level non-determinism may be small. Specifically, it took one of us (Zheng) less than one day to adapt the proxies designed for the auction to the bookstore. This is particularly promising in that he had no prior knowledge of the bookstore whatsoever.

One may argue that implementing JustRunIt may require a comparable amount of effort to developing accurate models for a service. We have experience with modeling the performance, energy, and temperature of server clusters and storage systems [4, 13, 12, 19] and largely agree with this claim. However, we note that Jus-

tRunIt is much more reusable than models, across different services, hardware and software characteristics, and even as service behavior evolves. Each of these factors requires model re-calibration and re-validation, which are typically labor-intensive. Furthermore, for models to become tractable, many simplifying assumptions about system behavior (e.g., memoryless request arrivals) may have to be made. These assumptions may compromise the accuracy of the models. JustRunIt does not require these assumptions and produces accurate results.

3 Experiment-based Management

As mentioned in the previous section, our infrastructure can be used by automated management systems or directly by the system administrator. To demonstrate its use in the former scenario, we have implemented simple automated management systems for two common tasks in virtualized hosting centers: server consolidation/expansion (i.e., partitioning resources across the services to use as few active servers as possible) and evaluation of hardware upgrades. These tasks are currently performed by most administrators in a manual, labor-intensive, and ad-hoc manner.

Both management systems seek to satisfy the services' SLAs. An SLA often specifies a percentage of requests to be serviced within some amount of time. Another possibility is for the SLA to specify an average response time (over a period of several minutes) for the corresponding service. For simplicity, our automated systems assume the latter type of SLA.

The next two subsections describe the management systems. However, before describing them, we note that they are *not* contributions of this work. Rather, they are presented simply to demonstrate the automated use of JustRunIt. More sophisticated systems (or the administrator) would leverage JustRunIt in similar ways.

3.1 Case Study 1: Resource Management

Overview. The ultimate goal of our resource-management system is to consolidate the hosted services onto the smallest possible set of nodes, while satisfying all SLAs. To achieve this goal, the system constantly monitors the average response time of each service, comparing this average to the corresponding SLA. Because workload conditions change over time, the resources assigned to a service may become insufficient and the service may start violating its SLA. Whenever such a violation occurs, our system initiates experiments with JustRunIt to determine what is the minimum allocation of resources that would be required for the service's SLA to be satisfied again. Changes in workload behavior often occur at the granularity of tens of minutes or even

```

1. While 1 do
2.   Monitor QoS of all services
3.   If any service needs more resources or
4.       can use fewer resources
5.     Run experiments with bottleneck tier
6.     Find minimum resource needs
7.     If used any interpolated results
8.       Inform JustRunIt about them
9.     Assign resources using bin-packing heuristic
10.    If new nodes need to be added
11.      Add new nodes and migrate VMs to them
12.    Else if nodes can be removed
13.      Migrate VMs and remove nodes
14.    Complete resource adjustments and migrations

```

Figure 4: Overview of resource-management system.

hours, suggesting that the time spent performing experiments is likely to be relatively small. Nevertheless, to avoid having to perform adjustments too frequently, the system assigns 20% more resources to a service than its minimum needs. This slack allows for transient increases in offered load without excessive resource waste. Since the resources required by the service have to be allocated to it, the new resource allocation may require VM migrations or even the use of extra nodes.

Conversely, when the SLA of any service is being satisfied by more than a threshold amount (i.e., the average response time is lower than that specified by the SLA by more than a threshold percentage), our system considers the possibility of reducing the amount of resources dedicated to the service. It does so by initiating experiments with JustRunIt to determine the minimum allocation of resources that would still satisfy the service's SLA. Again, we give the service additional slack in its resource allocation to avoid frequent reallocations. Because resources can be taken away from this service, the new combined resource needs of the services may not require as many PMs. In this case, the system determines the minimum number of PMs that can be used and implements the required VM migrations.

Details. Figure 4 presents pseudo-code overviewing the operation of our management system. The experiments with JustRunIt are performed in line 5. The management system only runs experiments with one software server of the bottleneck tier of the service in question. The management system can determine the bottleneck tier by inspecting the resource utilization of the servers in each tier. Experimenting with one software server is typically enough for two reasons: (1) services typically balance the load evenly across the servers of each tier; and (2) the VMs of all software servers of the same tier and service are assigned the same amount of resources at their PMs. (When at least one of these two properties does not hold, the management system needs to request more experiments of JustRunIt.) However, if enough nodes can be used for experiments in the sandbox, the system could run experiments with one software server from each tier of the service at the same time.

The matrix of resource allocations vs. response times produced by JustRunIt is then used to find the minimum resource needs of the service in line 6. Specifically, the management system checks the results in the JustRunIt matrix (from smallest to largest resource allocation) to find the minimum allocation that would still allow the SLA to be satisfied. In lines 7 and 8, the system informs JustRunIt about any interpolated results that it may have used in determining the minimum resource needs. JustRunIt will inform the management system if the interpolated results are different than the actual experimental results by more than a configurable threshold amount.

In line 9, the system executes a resource assignment algorithm that will determine the VM to PM assignment for all VMs of all services. We model resource assignment as a bin-packing problem. In bin-packing, the goal is to place a number of objects into bins, so that we minimize the number of bins. We model the VMs (and their resource requirements) as the objects and the PMs (and their available resources) as the bins. If more than one VM to PM assignment leads to the minimum number of PMs, we break the tie by selecting the optimal assignment that requires the smallest number of migrations. If more than one assignment requires the smallest number of migrations, we pick the one of these assignments randomly. Unfortunately, the bin-packing problem is NP-complete, so it can take an inordinate amount of time to solve it optimally, even for hosting centers of moderate size. Thus, we resort to a heuristic approach, namely simulated annealing [14], to solve it.

Finally, in lines 10–14, the resource-allocation system adjusts the number of PMs and the VM to PM assignment as determined by the best solution ever seen by simulated annealing.

Comparison. A model-based implementation for this management system would be similar; it would simply replace lines 5–8 with a call to a performance model solver. Obviously, the model would have to have been created, calibrated, and validated *a priori*.

A feedback-based implementation would replace lines 5–8 by a call to the controller to execute the experiments that will adjust the offending service. However, note that feedback control is only applicable when repeatedly varying the allocation of a resource or changing a hardware setting does not affect the on-line behavior of the co-located services. For example, we can use feedback control to vary the CPU allocation of a service without affecting other services. In contrast, increasing the amount of memory allocated to a service may require decreasing the allocation of another service. Similarly, varying the voltage setting for a service affects all services running on the same CPU chip, because the cores in current chips share the same voltage rail. Cross-service interactions are clearly undesirable, especially when they

1. For each service do
2. For one software server of each tier
3. Run experiments with JustRunIt
4. Find minimum resource needs
5. If used any interpolated results
6. Inform JustRunIt about them
7. Assign resources using bin-packing heuristic
8. Estimate power consumption

Figure 5: Overview of update-evaluation system.

may occur repeatedly as in feedback control. The key problem is that feedback control experiments with the on-line system. With JustRunIt, bin-packing and node addition/removal occur before any resource changes are made on-line, so interference can be completely avoided in most cases. When interference is unavoidable, e.g. the offending service cannot be migrated to a node with enough available memory and no extra nodes can be added, changes to the service are made only once.

3.2 Case Study 2: Hardware Upgrades

Overview. For our second case study, we built a management system to evaluate hardware upgrades. The system assumes that at least one instance of the hardware being considered is available for experimentation in the sandbox. For example, consider a scenario in which the hosting center is considering purchasing machines of a model that is faster or has more available resources than that of its current machines. After performing experiments with a single machine of the candidate model, our system determines whether the upgrade would allow servers to be consolidated onto a smaller number of machines and whether the overall power consumption of the hosting center would be smaller than it currently is. This information is provided to the administrator, who can make a final decision on whether or not to purchase the new machines and ultimately perform the upgrade.

Details. Figure 5 presents pseudo-code overviewing our update-evaluation system. The experiments with JustRunIt are started in line 3. For this system, the matrix that JustRunIt produces must include information about the average response time and the average power consumption of each resource allocation on the upgrade-candidate machine. In line 4, the system determines the resource allocation that achieves the same average response time as on the current machine (thus guaranteeing that the SLA would be satisfied by the candidate machine as well). Again, the administrator configures the system to properly drive JustRunIt and gets informed about any interpolated results that are used in line 4.

By adding the extra 20% slack to these minimum requirements and running the bin-packing algorithm described above, the system determines how many new machines would be required to achieve the current performance and how much power the entire center would

consume. Specifically, the center power can be estimated by adding up the power consumption of each PM in the resource assignment produced by the simulated annealing. The consumption of each PM can be estimated by first determining the “base” power of the candidate machine, i.e. the power consumption when the machine is on but no VM is running on it. This base power should be subtracted from the results in the JustRunIt matrix of each software server VM. This subtraction produces the average dynamic power required by the VM. Estimating the power of each PM then involves adding up the dynamic power consumption of the VMs that would run on the PM plus the base power.

Comparison. Modeling has been used for this management task [7]. A modeling-based implementation for our management system would replace lines 2–6 in Figure 5 with a call to a performance model solver to estimate the minimum resource requirements for each service. Based on these results and on the resource assignment computed in line 7, an energy model would estimate the energy consumption in line 8. Again, both models would have to have been created, calibrated, and validated *a priori*. In contrast, feedback control is not applicable to this management task.

4 Evaluation

4.1 Methodology

Our hardware comprises 15 HP Proliant C-class blades interconnected by a Gigabit Ethernet switch. Each server has 8 GBytes of DRAM, 2 hard disks, and 2 Intel dual-core Xeon CPUs. Each CPU has two frequency points, 2 GHz and 3 GHz. Two blades with direct-attached disks are used as network-attached storage servers. They export Linux LVM logical volumes to the other blades using ATA over Ethernet. One Gbit Ethernet port of every blade is used exclusively for network storage traffic. We measure the energy consumed by a blade by querying its management processor, which monitors the peak and average power usage of the entire blade.

Virtualization is provided by XenLinux kernel 2.6.18 with the Xen VMM [3], version 3.3. For improving Xen’s ability to provide performance isolation, we pin Dom0 to one of the cores and isolate the service(s) from it. Note, however, that JustRunIt does not itself impose this organization. As JustRunIt only depends on the VMM for VM cloning, it can easily be ported to use VMMs that do not perform I/O in a separate VM.

We populate the blade cluster with one or more independent instances of an on-line auction service. To demonstrate the generality of our system, we also experiment with an on-line bookstore. Both services are organized into three tiers of servers: Web, application, and

database tiers. The first tier is implemented by Apache Web servers (version 2.0.54), the second tier uses Tomcat servlet servers (version 4.1.18), and the third tier uses the MySQL relational database (version 5.0.27). (For performance reasons, the database servers are not virtualized and run directly on Linux and the underlying hardware.) We use LVS load balancers [30] in front of the Web and application tiers. The service requests are received by the Web servers and may flow towards the second and third tiers. The replies flow through the same path in the reverse direction.

We exercise each instance of the services using a client emulator. The auction workload consists of a “bidding mix” of requests (94% of the database requests are reads) issued by a number of concurrent clients that repeatedly open sessions with the service. The bookstore workload comprises a “shopping mix”, where 20% of the requests are read-write. Each client issues a request, receives and parses the reply, “thinks” for a while, and follows a link contained in the reply. A user-defined Markov model determines which link to follow. The code for the services, their workloads, and the client emulator are from the DynaServer project [20] and have been used extensively by other research groups.

4.2 JustRunIt Overhead

Our overhead evaluation seeks to answer two questions: (1) Does the overhead of JustRunIt (proxies, VM cloning, workload duplication, and reply matching) degrade the performance of the on-line services? and (2) How faithfully do servers in the sandbox represent on-line servers given the same resources?

To answer these questions, we use our auction service as implemented by one Apache VM, one Tomcat VM, and MySQL. Using a larger instance of the service would hide some of the overhead of JustRunIt, since the proxies only instrument one path through the service. Each of the VMs runs on a different blade. We use one blade in the sandbox. The two proxies for the Web tier run on one of the blades, whereas those for the application tier run on another. The proxies run on their own blades to promote performance isolation for the auction service. In all our experiments, the time shift used by JustRunIt is 10 seconds behind the on-line service.

Overhead on the on-line system? To isolate the overhead of JustRunIt on the on-line service, we experiment with three scenarios: (1) Plain – no proxies are installed; (2) ProxiesInstalled – proxies are installed around the Web and application servers, but they only relay the network traffic; and (3) JustRunIt – proxies are installed around the Web and application servers and perform all the JustRunIt functionality.

Figures 6 and 7 depict the average throughput and re-

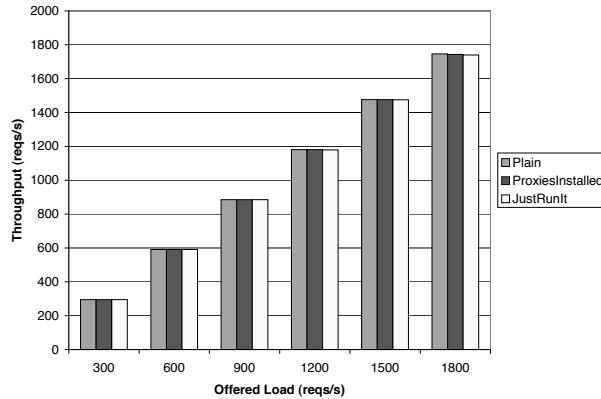


Figure 6: Throughput as a function of offered load.

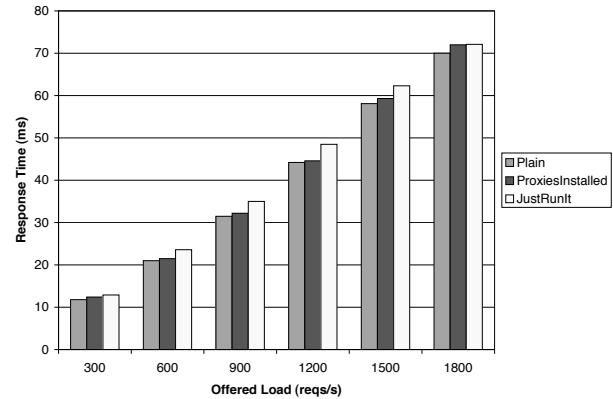


Figure 7: Response time as a function of offered load.

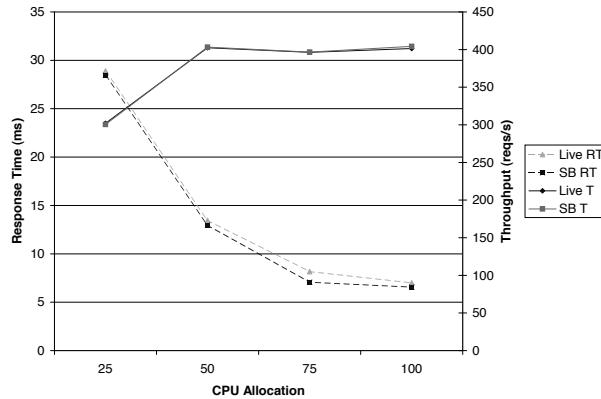


Figure 8: On-line and sandboxed performance as a function of CPU allocation at offered load 500 requests/second.

sponse time of the on-line service, respectively, as a function of the offered load. We set the CPU allocation of all servers to 100% of one core. In this configuration, the service saturates at 1940 requests/second. Each bar corresponds to a 200-second execution.

Figure 6 shows that JustRunIt has no effect on the throughput of the on-line service, even as it approaches saturation, despite having the proxies for each tier co-located on the same blade.

Figure 7 shows that the overhead of JustRunIt is consistently small (< 5ms) across load intensities. We are currently in the process of optimizing the implementation to reduce the JustRunIt overheads further. However, remember that the overheads in Figure 7 are exaggerated by the fact that, in these experiments, *all* application server requests are exposed to the JustRunIt instrumentation. If we had used a service with 4 application servers, for example, only roughly 25% of those requests would be exposed to the instrumentation (since we only need proxies for 1 of the application servers), thus lowering the average overhead by 75%.

Performance in the sandbox? The results above isolate

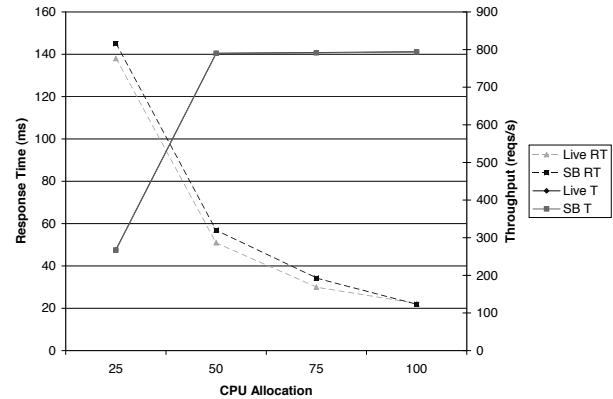


Figure 9: On-line and sandboxed performance as a function of CPU allocation at offered load 1000 requests/second.

the overhead of JustRunIt on the on-line system. However, another important consideration is how faithful the sandbox execution is to the on-line execution given the same resources. Obviously, it would be inaccurate to make management decisions based on sandboxed experiments that are not very similar to the behavior of the on-line system.

Figures 8 and 9 compare the performance of the on-line application server (labeled “Live”) to that of the sandboxed (labeled “SB”) application server at 500 requests/second and 1000 requests/second, respectively. In both figures, response times (labeled “RT”) and throughputs (labeled “T”) are measured at the application server’s in-proxy. Again, each result represents the average performance over 200 seconds.

As one would expect, the figures show that increasing the CPU allocation tends to increase throughputs and reduce response times. The difference between the offered load and the achieved throughput is the 20% of requests that are served directly by the Web server and, thus, do not reach the application server’s in-proxy. More interestingly, the figures clearly show that the sandboxed execution is a faithful representation of the on-line system,

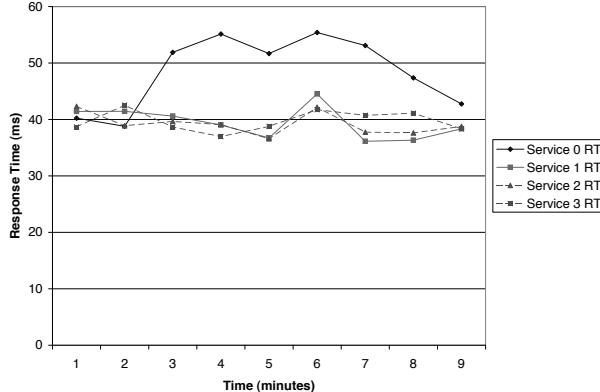


Figure 10: Server expansion using JustRunIt.

regardless of the offered load.

The results for the Web tier also show the sandboxed execution to be accurate. Like the application-tier results, we ran experiments with four different CPU allocations, under two offered loads. When the offered load is 500 reqs/s, the average difference between the on-line and sandboxed results is 4 requests/second for throughput and 1 ms for response time, across all CPU allocations. Even under a load of 1000 requests/second, the average throughput and response time differences are only 6 requests/second and 2 ms, respectively.

Our experiments with the bookstore service exhibit the same behaviors as in Figures 6 to 9. The throughput is not affected by JustRunIt and the overhead on the response time is small. For example, under an offered load of 300 requests/second, JustRunIt increases the mean response time for the bookstore from 18 ms to 22 ms. For 900 requests/second, the increase is from 54 ms to 58 ms. Finally, our worst result shows that JustRunIt increases the mean response time from 90 ms to 100 ms at 1500 requests/second.

4.3 Case Study 1: Resource Management

As mentioned before, we built an automated resource manager for a virtualized hosting center that leverages JustRunIt. To demonstrate the behavior of our manager, we created four instances of our auction service on 9 blades: 2 blades for first-tier servers, 2 blades for second-tier servers, 2 blades for database servers, and 3 blades for storage servers and LVS. Each first-tier (second-tier) blade runs one Web (application) server from each service. Each server VM is allocated 50% of one core as its CPU allocation. We assume that the services' SLAs require an average response time lower than 50 ms in every period of one minute. The manager requested JustRunIt to run 3 CPU-allocation experiments with any service that violated its SLA, for no longer than 3 minutes overall. A 10th blade is used for the JustRunIt sand-

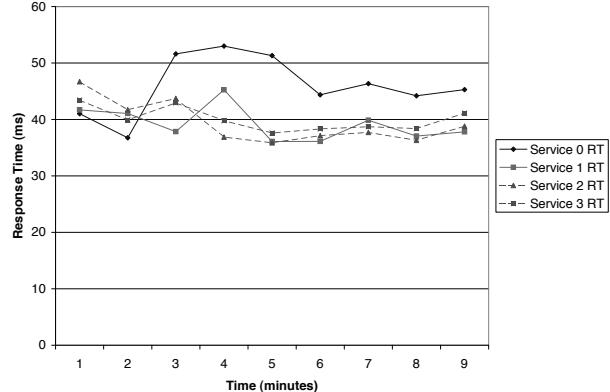


Figure 11: Server expansion using accurate modeling.

box, whereas 2 extra blades are used for its Web and application-server proxies. Finally, 2 more blades are used to generate load.

Figure 10 shows the response time of each service during our experiment; each point represents the average response time during the corresponding minute. We initially offered 1000 requests/second to each service. This offered load results in an average response time hovering around 40 ms. Two minutes after the start of the experiment, we increase the load offered to service 0 to 1500 requests/second. This caused its response time to increase beyond 50 ms during the third minute of the experiment. At that point, the manager started JustRunIt experiments to determine the CPU allocation that would be required for the service's application servers (the second tier is the bottleneck tier) to bring response time back below 50 ms under the new offered load. The set of JustRunIt experiments lasted 3 minutes, allowing CPU allocations of 60%, 80%, and 100% of a core to be tested. The values for 70% and 90% shares were interpolated based on the experimental results.

Based on the response-time results of the experiments, the manager determined that the application server VMs of the offending service should be given 72% of a core (i.e., 60% of a core plus the 20% of 60% = 12% slack). Because of the extra CPU allocation requirements, the manager decided that the system should be expanded to include an additional PM (a 15th blade in our setup). To populate this machine, the manager migrated 2 VMs to it (one from each PM hosting application server VMs). Besides the 3 minutes spent with experiments, VM cloning, simulated annealing, and VM migration took about 1 minute altogether. As a result, the manager was able to complete the resource reallocation 7 minutes into the experiment. The experiment ended with all services satisfying their SLAs.

Comparison against highly accurate modeling. Figure 11 shows what the system behavior would be if the

resource manager made its decisions based on a highly accurate response-time model of our 3-tier auction service. To mimic such a model, we performed the JustRunIt experiments with service 0 under the same offered load of Figure 10 for all CPU allocations off-line. These off-line results were fed to the manager during the experiment free of any overheads. We assumed that the model-based manager would require 1 minute of resource-usage monitoring after the SLA violation is detected, before the model could be solved. Based on the JustRunIt results, the manager made the same decisions as in Figure 10.

The figure shows that modeling would allow the system to adjust 2 minutes faster. However, developing, calibrating, and validating such an accurate model is a challenging and labor-intensive proposition. Furthermore, adaptations would happen relatively infrequently in practice, given that (1) it typically takes at least tens of minutes for load intensities to increase significantly in real systems, and (2) the manager builds slack into the resource allocation during each adaptation. In summary, the small delay in decision making and the limited resources that JustRunIt requires are a small price to pay for the benefits that it affords.

4.4 Case Study 2: Hardware Upgrade

We also experimented with our automated system for evaluating hardware upgrades in a virtualized hosting center. To demonstrate the behavior of our system, we ran two instances of our auction service on the same number of blades as in our resource manager study above. However, we now configure the blades that run the services to run at 2 GHz. The blade in the JustRunIt sandbox is set to run at 3 GHz to mimic a more powerful machine that we are considering for an upgrade of the data center. We offer 1000 requests/second to each service. We also cap each application server VM of both services at 90% of one core; for simplicity, we do not experiment with the Web tier, but the same approach could be trivially taken for it as well.

During the experiment, the management system requested JustRunIt to run 4 CPU-allocation experiments for no longer than 800 seconds overall. (Note, though, that this type of management task does not have real-time requirements, so we can afford to run JustRunIt experiments for a much longer time.) Since each server is initially allocated 90% of one core, JustRunIt is told to experiment with CPU allocations of 50%, 60%, 70%, and 80% of one core; there is no need for interpolation. The management system's main goal is to determine (using simulated annealing) how many of the new machines would be needed to achieve the same response time that the services currently exhibit. With this information, the energy implications of the upgrade can be assessed.

Based on the results generated by JustRunIt, the management system decided that the VMs of both services could each run at 72% CPU allocations (60% of one core plus 12% slack) at 3 GHz. For a large data center with diverse services, a similar reduction in resource requirements may allow for servers to be consolidated, which would most likely conserve energy. Unfortunately, our experimental system is too small to demonstrate these effects here.

4.5 Summary

In summary, the results above demonstrate that the JustRunIt overhead is small, even when all requests are exposed to our instrumentation. In real deployments, the observed overhead will be even smaller, since there will certainly be more than one path through each service (at the very least to guarantee availability and fault-tolerance). Furthermore, the results show that the sandboxed execution is faithful to the on-line execution. Finally, the results demonstrate that JustRunIt can be effectively leveraged to implement sophisticated automated management systems. Modeling could have been applied to the two systems, whereas feedback control is applicable to resource management (in the case of the CPU allocation), but not upgrade evaluation. The hardware resources consumed by JustRunIt amount to one machine for the two proxies of each tier, plus as few as one sandbox machine. Most importantly, *this overhead is fixed and independent of the size of the production system*.

5 Related Work

Modeling, feedback control, and machine learning for managing data centers. State-of-the-art management systems rely on analytical modeling, feedback control, and/or machine learning to at least partially automate certain management tasks. As we have mentioned before, modeling has complexity and accuracy limitations, whereas feedback control is not applicable to many types of tasks. Although machine learning is useful for certain management tasks, such as fault diagnosis, it also has applicability limitations. The problem is that machine learning can only learn about system scenarios and configurations that have been seen in the past and about which enough data has been collected. For example, it applies to neither of the tasks we study in this paper. Nevertheless, machine learning can be used to improve the interpolation done by JustRunIt, when enough data exists for it to derive accurate models.

JustRunIt takes a fundamentally different approach to management; one in which accurate sandboxed experiments replace modeling, feedback control, and machine learning.

Scaling down data centers. Gupta *et al.* [10] proposed the DieCast approach for scaling down a service. DieCast enables some management tasks, such as predicting service performance as a function of workload, to be performed on the scaled version. Scaling is accomplished by creating one VM for each PM of the service and running the VMs on an off-line cluster that is an order of magnitude smaller than the on-line cluster. Because of the significant scaling in size, DieCast also uses time dilation [11] to make guest OSes think that they are running on much faster machines. For a 10-fold scale down, time dilation extends execution time by 10-fold.

DieCast and JustRunIt have fundamentally different goals and resource requirements. First, JustRunIt targets a subset of the management tasks that DieCast does; the subset that can be accomplished with limited additional hardware resources, software infrastructure, and costs. In particular, JustRunIt seeks to improve upon modeling by leveraging native execution. Because of time dilation, DieCast takes excessively long to perform each experiment. Second, JustRunIt includes infrastructure for automatically experimenting with services, as well as interpolating and checking the experimental results. Third, JustRunIt minimizes the set of hardware resources that are required by each experiment without affecting its running time. In contrast, to affect execution time by a small factor, DieCast requires an additional hardware infrastructure that is only this same small factor smaller than the entire on-line service.

Sandboxing and duplication for managing data centers. A few efforts have proposed related infrastructures for managing data centers. Specifically, [15, 17] considered validating operator actions in an Internet service by using request duplication to a sandboxed extension of the service. For each request, if the replies generated by the on-line environment and by the sandbox ever differ during a validation period, a potential operator mistake is flagged. Tan *et al.* [25] considered a similar infrastructure for verifying file servers.

Instead of operator-action validation in a single, non-virtualized Internet service, our goal is to experimentally evaluate the effect of different resource allocations, parameter settings, and other potential system changes (such as hardware upgrades) in virtualized data centers. Thus, JustRunIt is much more broadly applicable than previous works. As a result, our infrastructure is quite different than previous systems. Most significantly, JustRunIt is the first system that may explore a large number of scenarios that differ from the on-line system, while extrapolating results from the experiments that are actually run, and verifying its extrapolations if necessary.

Selecting experiments to run. Previous works have proposed sophisticated approaches for selecting the experi-

ments to run when benchmarking servers [22] or optimizing their configuration parameters [26, 31]. Such approaches are largely complementary to our work. Specifically, they can be used to improve experiment-based management in two ways: (1) automated management systems can use them to define/constrain the parameter space that JustRunIt should explore; or (2) they can be used as new heuristics in JustRunIt’s driver to eliminate unnecessary experiments.

6 Conclusions

This paper introduced a novel infrastructure for experiment-based management of virtualized data centers, called JustRunIt. The infrastructure enables an automated management system or the system administrator to answer “what-if” questions experimentally during management tasks and, based on the answers, select the best course of action. The current version of JustRunIt can be applied to many management tasks, including resource management, hardware upgrades, and software upgrades.

Limitations. There are three types of “what-if” questions that sophisticated models can answer (by making simplifying assumptions and costing extensive human labor), whereas JustRunIt currently cannot. First, service-wide models can answer questions about the effect of a service tier on other tiers. In the current version of JustRunIt, these cross-tier interactions are not visible, since the sandboxed virtual machines do not communicate with each other.

Second, models that represent request mixes at a low enough level can answer questions about hypothetical mixes that have not been experienced in practice. Currently, JustRunIt relies solely on real workload duplication for its experiments, so it can only answer questions about request mixes that are offered to the system. Nevertheless, JustRunIt *can* currently answer questions about more or less intense versions of real workloads, which seems to be a more useful property.

Finally, models can sometimes be used to spot performance anomalies, although differences between model results and on-line behavior are often due to inaccuracies of the model. Because JustRunIt uses complete-state replicas of on-line virtual machines for greater realism in its experiments, anomalies due to software server or operating system bugs cannot be detected.

Future work. We plan to extend JustRunIt to allow cross-tier communication between the sandboxed servers. This will allow the administrator to configure sandboxing with or without cross-tier interactions. We also plan to create infrastructure to allow experimentation with request mixes other than those observed on-

line. The idea here is to collect a trace of the on-line workload offered to one server of each tier, as well as the state of these servers. Later, JustRunIt could install the states and replay the trace to the sandboxed servers. During replay, the request mix could be changed by eliminating or replicating some of the traced sessions. Finally, we plan to build an in-proxy for a database server, starting with code from the C-JDBC middleware.

Acknowledgements

We would like to thank our shepherd, John Dunagan, and the anonymous reviewers for comments that helped improve this paper significantly. This research was partially supported by Hewlett-Packard and by NSF grant #CSR-0509007.

References

- [1] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-Based Network Servers. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2000.
- [2] Banu. Tinyproxy. <http://www.banu.com/tinyproxy/>, 2008.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.
- [4] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini. User-Level Communication in Cluster-Based Servers. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, February 2002.
- [5] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, June 2004.
- [6] J. Chase, D. Anderson, P. Thackar, A. Vahdat, and R. Boyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th Symposium on Operating Systems Principles*, October 2001.
- [7] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing Server Energy and Operational Costs in Hosting Centers. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2005.
- [8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the International Symposium on Networked Systems Design and Implementation*, 2005.
- [9] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.
- [10] D. Gupta, K. Vishwanath, and A. Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proceedings of the International Symposium on Networked Systems Design and Implementation*, May 2008.
- [11] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, G. M. Voelker, and A. Vahdat. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the International Symposium on Networked Systems Design and Implementation*, May 2006.
- [12] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and Freon: Temperature Emulation and Management for Server Systems. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.
- [13] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr., and R. Bianchini. Energy Conservation in Heterogeneous Server Clusters. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.
- [14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), 1983.
- [15] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, December 2004.
- [16] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, April 2005.
- [17] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Validating Database System Administration. In *Proceedings of USENIX Annual Technical Conference 2006*, June 2006.
- [18] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of EuroSys*, March 2007.
- [19] E. Pinheiro, R. Bianchini, and C. Dubnicki. Exploiting Redundancy to Conserve Energy in Storage Systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2006.
- [20] Rice University. DynaServer Project. <http://www.cs.rice.edu/CS/Systems/DynaServer>, 2003.
- [21] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

- [22] P. Shivam, V. Marupadi, J. Chase, and S. Babu. Cutting Corners: Workbench Automation for Server Benchmarking. In *Proceedings of the 2008 USENIX Annual Technical Conference*, June 2008.
- [23] C. Stewart, T. Kelly, A. Zhang, and K. Shen. A Dollar from 15 Cents: Cross-Platform Management for Internet Services. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.
- [24] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proceedings of the International Symposium on Networked Systems Design and Implementation*, May 2005.
- [25] Y.-L. Tan, T. Wong, J. D. Strunk, and G. R. Ganger. Comparison-based File Server Verification. In *Proceedings of the USENIX Annual Technical Conference*, June 2005.
- [26] R. Thonangi, V. Thummala, and S. Babu. Finding Good Configurations in High-Dimensional Spaces: Doing More with Less. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, September 2008.
- [27] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile Dynamic Provisioning of Multi-tier Internet Applications. *ACM Transactions on Adaptive and Autonomous Systems*, 3(1), March 2008.
- [28] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.
- [29] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousef. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, April 2007.
- [30] W. Zhang. Linux Virtual Server for Scalable Network Services. In *Proceedings of the Linux Symposium*, July 2000.
- [31] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic Configuration of Internet Services. In *Proceedings of Eurosys*, March 2007.

vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities

Byung Chul Tak*, Chunqiang Tang†, Chun Zhang†,
Sriram Govindan*, Bhuvan Urgaonkar*, and Rong N. Chang†

*Dept. of Computer Science and Engineering, Pennsylvania State University

†IBM T.J. Watson Research Center

Abstract

Discovering end-to-end request-processing paths is crucial in many modern IT environments for reasons varying from debugging and bottleneck analysis to billing and auditing. Existing solutions for this problem fall into two broad categories: statistical inference and intrusive instrumentation. The statistical approaches infer request-processing paths in a “most likely” way and their accuracy degrades as the workload increases. The instrumentation approaches can be accurate, but they are system dependent as they require knowledge (and often source code) of the application as well as time and effort from skilled programmers.

We have developed a discovery technique called *vPath* that overcomes these shortcomings. Unlike techniques using statistical inference, *vPath* provides precise path discovery, by monitoring thread and network activities and reasoning about their causality. Unlike techniques using intrusive instrumentation, *vPath* is implemented in a virtual machine monitor, making it agnostic of the underlying middleware or application. Our evaluation using a diverse set of applications (TPC-W, RUBiS, MediaWiki, and the home-grown *vApp*) written in different programming languages (C, Java, and PHP) demonstrates the generality and accuracy of *vPath* as well as its low overhead. For example, turning on *vPath* affects the throughput and response time of TPC-W by only 6%.

1 Introduction

The increasing complexity of IT systems is well documented [3, 8, 28]. As a legacy system evolves over time, existing software may be upgraded, new applications and hardware may be added, and server allocations may be changed. A complex IT system typically includes hardware and software from multiple vendors. Administrators often struggle with the complexity of and pace of changes to their systems.

This problem is further exacerbated by the much-touted IT system “agility,” including dynamic application placement [29], live migration of virtual ma-

chines [10], and flexible software composition through Service-Oriented Architecture (SOA) [11]. Agility promotes the value of IT, but makes it even harder to know *exactly* how a user request travels through distributed IT components. For instance, was server *X* in a cluster actually involved in processing a given request? Was a failure caused by component *Y* or *Z*? How many database queries were used to form a response? How much time was spent on each involved component? Lack of visibility into the system can be a major obstacle for accurate problem determination, capacity planning, billing, and auditing.

We use the term, *request-processing path*, to represent all activities starting from when a user request is received at the front tier, until the final response is sent back to the user. A request-processing path may comprise multiple messages exchanged between distributed software components, e.g., Web server, LDAP server, J2EE server, and database. Understanding the request-processing path and the performance characteristics of each step along the path has been identified as a crucial problem. Existing solutions for this problem fall into two broad categories: intrusive instrumentation [4, 20, 9, 8, 30] and statistical inference [1, 21, 3, 32, 25].

The instrumentation-based approaches are *precise* but *not general*. They modify middleware or applications to record events (e.g., request messages and their end-to-end identifiers) that can be used to reconstruct request-processing paths. Their applicability is limited, because it requires knowledge (and often source code) of the specific middleware or applications in order to do instrumentation. This is especially challenging for complex IT systems that comprise middleware and applications from multiple vendors.

Statistical approaches are *general* but *not precise*. They take readily available information (e.g., timestamps of network packets) as inputs, and infer request-processing paths in a “most likely” way. Their accuracy degrades as the workload increases, because of the difficulty in differentiating activities of concurrent requests. For example, suppose a small fraction of requests have

strikingly long response time. It would be helpful to know exactly how a slow request and a normal request differ in their processing paths—which servers they visited and where the time was spent. However, the statistical approaches cannot provide *precise* answers for individual requests.

The IBM authors on this paper build tools for and directly participate in consulting services [13] that help customers (e.g., commercial banks) diagnose problems with their IT systems. In the past, we have implemented tools based on both statistical inference [32] and application/middleware instrumentation. Motivated by the challenges we encountered in the field, we set out to explore whether it is possible to design a request-processing path discovery method that is both *precise* and *general*. It turns out that this is actually doable for most of the commonly used middleware and applications.

Our key observation is that most distributed systems follow two fundamental programming patterns: (1) *communication pattern*—synchronous request-reply communication (i.e., synchronous RPC) over TCP connections, and (2) *thread pattern*—assigning a thread to do most of the processing for an incoming request. These patterns allow us to precisely reason about event causality and reconstruct request-processing paths without system-dependent instrumentation. Specifically, the thread pattern allows us to infer causality within a software component, i.e., processing an incoming message X triggers sending an outgoing message Y . The communication pattern allows us to infer causality between two components, i.e., application-level message Y sent by one component corresponds to message Y' received by another component. Together, knowledge of these two types of causality helps us to precisely reconstruct end-to-end request-processing paths.

Following these observations, our technique reconstructs request-processing paths from minimal information recorded at runtime—which thread performs a send or `recv` system call over which TCP connection. It neither records message contents nor tracks end-to-end message identifiers. Our method can be implemented efficiently in either the OS kernel or a virtual machine monitor (VMM). Finally, it is completely agnostic to user-space code, thereby enabling accurate discovery of request-processing paths for most of the commonly used middleware and applications.

In general, a VMM-based implementation of our method is more challenging than an OS-based implementation, because it is more difficult to obtain thread and TCP information in a VMM. This paper presents a VMM-based implementation, because we consider it easier to deploy such a solution in cloud-computing environments such as Amazon’s EC2 [2]. Our implementation is based on Xen [5]. In addition to modifying the VMM code, our current prototype still makes minor

changes to the guest OS. We will convert it to a pure VMM-based implementation after the ongoing fast prototyping phase.

1.1 Research Contributions

We propose a novel set of techniques called *vPath*, for discovering end-to-end request-processing paths, which addresses most of the shortcomings of existing approaches. Specifically, we make the following contributions:

- *New angle for solving a well-known problem*: Most recent work focused on developing better statistical inference models or different application instrumentation techniques. We instead take a very different angle—exploiting common programming patterns—to radically simplify the problem.
- *Implementation and generality*: We implement vPath by modifying Xen, without modifying any user-space code. Although vPath makes certain assumptions about the application’s programming patterns (synchronous remote invocation and causality of thread activities), we argue and corroborate from experiments and existing literature, that this does not diminish the general applicability of vPath.
- *Completeness and accuracy*: We conduct an extensive evaluation of vPath, using a diverse set of applications (TPC-W, RUBiS, MediaWiki, and the home-grown vApp) written in different languages (C, Java, and PHP). Our experiments demonstrate vPath’s completeness (ability to discover all request paths), accuracy (all discovered request paths are correct), and efficiency (negligible impact on overlying applications).

The rest of this paper is organized as follows. Section 2 presents an overview of vPath. Section 3 describes vPath’s implementation in detail. In Section 4, we empirically evaluate various aspects of vPath. We discuss related work in Section 5, and present concluding remarks in Section 6.

2 Overview of vPath

In this section, we present an overview of vPath and discuss its applicability to existing software architectures.

2.1 Goodness Criteria

Several criteria are meaningful in assessing the desirability and efficacy of any request path discovery technique. Our design of vPath takes the following five into consideration. The first three are quantifiable metrics, while the last two are subjective.

- **Completeness** is the ratio of correctly discovered request paths to all paths that actually exist.
- **Accuracy** is the ratio of correctly discovered request paths to all paths reported by a technique.

- **Efficiency** measures the runtime overhead that a discovery technique imposes on the application.
- **Generality** refers to the hardware/software configurations to which a discovery technique is applicable, including factors such as programming language, software stack (e.g., one uniform middleware or heterogeneous platforms), clock synchronization, presence or absence of application-level logs, communication pattern, threading model, to name a few.
- **Transparency** captures the ability to avoid understanding or changing user-space code. We opt for changing OS kernel or VMM, because it only needs to be done once. By contrast, a user-space solution needs intrusive modifications to every middleware or application written in every programming language.

2.2 Assumptions Made by vPath

vPath makes certain assumptions about a distributed system’s programming pattern. We will show that these assumptions hold for many commonly used middleware and applications. vPath assumes that (1) distributed components communicate through synchronous request-reply messages (i.e., synchronous RPC), and (2) inside one component, causality of events is observable through thread activities.

Communication-pattern assumption. With the synchronous communication pattern, a thread in one component sends a request message over a TCP connection to a remote component, and then *blocks* until the corresponding reply message comes back over the same TCP connection. This implies that the second request may only be sent over the same TCP connection (by any thread) after receiving the reply message for the first request.

Thread-pattern assumption. Suppose an incoming request X (e.g., an HTTP request) to a software component triggers one or more subordinate requests Y (e.g., LDAP authentication and database queries) being sent to other components. Requests X and Y belong to the same request-processing path. vPath assumes that the thread that sends X ’s reply message back to the upstream component is also the thread that sends all the subordinate request messages Y to the downstream components. Moreover, this thread does not send messages on behalf of other user requests during that period of time.

Consider the example in Figure 1, where $\text{request-}X$ received by *component-I* triggers $\text{request-}Y$ being sent to *component-II*. vPath assumes that $\text{send-request-}Y$ and $\text{send-reply-}X$ are performed by the same thread. On the other hand, vPath allows that another thread (e.g., a front-end dispatcher thread) performs the $\text{recv-request-}X$ operation and then one or more threads perform some pre-processing on the request before the request is handed to the last thread in this processing chain for final handling. vPath

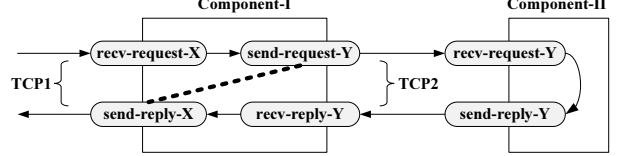


Figure 1: An example of a request-processing path. The rectangles (components I and II) represent distributed software components. The ellipses represent events observed at individual components, e.g., $\text{recv-request-}X$ is the event that message X -request is received by a thread in *component-I*. Message $\text{reply-}X$ is the response to message $\text{request-}X$. $\text{request-}X$ and $\text{reply-}X$ are sent over TCP1. $\text{request-}Y$ and $\text{reply-}Y$ are sent over TCP2. The arrows show the request-processing path. The dotted line shows the conceptual linkage between $\text{send-request-}Y$ and $\text{send-reply-}X$, which is the assumption of vPath, i.e., the same thread performs the two send operations.

only requires that this last thread performs both send operations ($\text{send-request-}Y$ and $\text{send-reply-}X$).

Our discussion above focused on only one request. vPath supports multiple threads in one component concurrently processing different requests. These threads can execute in any order dictated by the CPU scheduler and synchronization libraries, producing interleaved sequences of request messages and reply messages.

2.3 Discovering Request-Processing Paths with vPath

To reconstruct request-processing paths, vPath needs to infer two types of causality. *Intra-node causality* captures the behavior that, within one component, processing an incoming message X triggers sending an outgoing message Y . *Inter-node causality* captures the behavior that, an application-level message Y sent by one component corresponds to message Y' received by another component. Our thread-pattern assumption enables the inference of intra-node causality, while the communication-pattern assumption enables the inference of inter-node causality.

Specifically, vPath reconstructs the request-processing path in Figure 1 as follows. Inside *component-I*, the synchronous-communication assumption allows us to match the first incoming message over TCP1 with the first outgoing message over TCP1, match the second incoming message with the second outgoing message, and so forth. (Note that one application-level message may be transmitted as multiple network-level packets.) Therefore, $\text{recv-request-}X$ can be correctly matched with $\text{send-reply-}X$. Similarly, we can match *component-I*’s $\text{send-request-}Y$

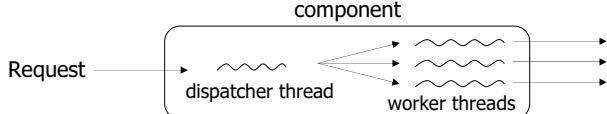


Figure 2: Dispatcher-worker threading model.

with `recv-reply-Y`, and also match *component-II*'s `recv-request-Y` with `send-reply-Y`.

Between two components, we can match *component-I*'s first outgoing message over TCP2 with *component-II*'s first incoming message over TCP2, and so forth, hence, correctly matching *component-I*'s `send-request-Y` with *component-II*'s `recv-request-Y`.

The only missing link is that, in *component-I*, `recv-request-X` triggers `send-request-Y`. From the thread-pattern assumption, we can indirectly infer this causality with the help of the dotted line in Figure 1. Recall that we have already matched `recv-request-X` with `send-reply-X`. Between the time of these two operations, we observe that the same thread performs `send-request-Y` and `send-reply-X`. It follows from our thread-pattern assumption that `recv-request-X` triggers `send-request-Y`. This completes the construction of the end-to-end execution path in Figure 1.

As described above, the amount of information needed by vPath to discover request-processing paths is very small. vPath only needs to monitor which thread performs a send or receive system call over which TCP connection. This information can be obtained efficiently in the OS kernel or VMM, without modifying any user-space code. Unlike existing methods [19, 30, 9], vPath needs neither message contents nor end-to-end message identifiers.

2.4 Applicability of vPath to Existing Threading Models

In this section, we summarize three well-known threading models, and discuss vPath's applicability and limitations with respect to these models. For a more detailed study and comparison of these models, we encourage readers to refer to [7, 18, 34].

2.4.1 Dispatcher-worker Threading Model

Figure 2 shows a component of an application built using the dispatcher-worker model, which is arguably the most widely used threading model for server applications. In the front-end, one or more dispatcher threads use the `select()` system call or the `accept()` system call to detect new incoming TCP connections or new requests over existing TCP connections. Once a request is identified, it is handed over to a worker thread for further processing. This single worker thread is responsible for executing all activities triggered by the request (e.g.,

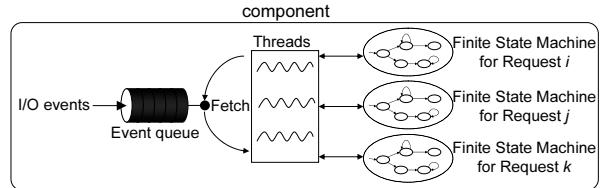


Figure 3: Event-driven model.

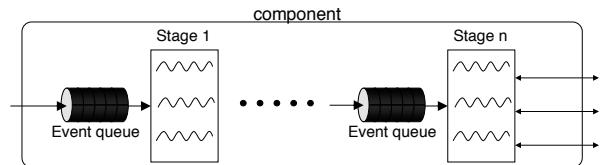


Figure 4: Staged Event-Driven Architecture.

reading HTML files from a disk or making JDBC calls to a database), and finally sending a reply message back to the user. After the worker thread finishes processing the request, it goes back into the worker thread pool, waiting to be picked to process another incoming request.

This threading model conforms to vPath's thread-pattern assumption described in Section 2.2. Since a single worker thread executes all activities triggered by a request, the worker thread performs both `send-request-Y` and `send-reply-X` in Figure 1.

2.4.2 Event-Driven Model

Figure 3 shows the architecture of an application's component built using the event-driven programming model. Compared with other threading models, the event-driven model uses a relatively small number of threads, typically equal to or slightly larger than the number of CPUs. When processing a request R , a thread T_1 always uses non-blocking system calls. If it cannot make progress on processing the request R because a non-blocking I/O operation on behalf of R has not yet completed, the thread T_1 records the current status of R in a finite state machine maintained for R , and moves on to process another request. When the I/O operation on behalf of R finishes, an event is created in the event queue, and eventually a thread T_2 retrieves the event and continues to process R . Note that T_1 and T_2 may be different threads, both participating in processing the same request at different times. The event-driven model does not conform to vPath's thread-pattern assumption, and cannot be handled by vPath.

2.4.3 Staged Event-Driven Architecture (SEDA) Model

Figure 4 shows the architecture of a SEDA-based application component [34]. SEDA partitions the request processing pipeline into stages and each stage has its

own thread pool. Any two neighboring stages are connected by an event queue. SEDA partially conforms to vPath’s assumptions. If only the last stage sends outgoing messages, and if communication between distributed components is synchronous (as described in Section 2.2), then vPath will be able to correctly discover request-processing paths. Otherwise, vPath would fail.

2.5 Why vPath is Still Useful

Among the three well-known threading models, vPath can handle the dispatcher-worker thread model, only partially handles the SEDA model, and cannot handle the event-driven model. However, we still consider vPath as a widely applicable and general solution, because the dispatcher-worker thread model is the dominant architecture among mainstream software. The wide adoption of the dispatcher-worker thread model is not accidental. Consider, for example, common middleware platforms such as J2EE, where threads are managed by the middleware and used to execute user code written by different programmers. Because the middleware cannot make strong assumptions about the user code’s behavior (e.g., blocking or not), it is simplest and safest to adopt the dispatcher-worker thread model.

The SEDA model has been widely discussed within the research community, but no consensus about its suitability has been reached (see Welsh’s discussion in [33]). Moreover, wide adoption of the SEDA model in mainstream software is yet to be reported.

The pure event-driven model in Figure 3 is rarely used in real applications. The Flash Web server [18] is often considered as a notable example that adopts the event-driven model, but Flash actually uses a hybrid between event-driven and multi-threaded programming models. In Flash, a single main thread does all non-blocking network I/O operations and a set of worker threads do blocking disk I/O operations. The event-driven model is not yet popular in real applications and there is considerable consensus in the research community that it is difficult to program and debug applications based on a pure event-driven model. Similar sentiments were expressed by Behren et al. [6], who have had extensive experience programming a variety of applications using the event-driven approach.

Furthermore, even the frequently-cited performance advantages of the event-driven model are questionable in practice, as it is extremely hard to ensure that a thread actually never blocks. For example, the designers of Flash themselves observed that the supposedly never-blocking main thread actually blocks unexpectedly in the “find file” stage of HTTP request processing, and subsequently published multiple research papers [22, 23] describing how they solved the problem by hacking the operating system. Considering the excellent expertise of the Flash researchers on this subject and the relatively small code

size of Flash, it is hard to imagine that ordinary programmers working on complex commercial software would have a better chance of getting the implementation right.

Because vPath’s assumptions hold for most of the existing mainstream software, we consider vPath as a widely applicable and general solution. In Section 4, we will validate this using a wide range of applications, written in different programming languages, developed by a variety of communities.

3 Implementation of vPath

The vPath toolset consists of an online monitor and an offline log analyzer. The online monitor continuously logs which thread performs a `send` or `recv` system call over which TCP connection. The offline log analyzer parses logs generated by the online monitor to discover request-processing paths and the performance characteristics at each step along these paths.

The online monitor tracks network-related thread activities. This information helps infer the intra-node causality of the form “processing an incoming message X triggers sending an outgoing message Y .” It also tracks the identity of each TCP connection, i.e., the four-element tuple $(source_IP, source_port, dest_IP, dest_port)$ that uniquely identifies a live TCP connection at any moment in time. This information helps infer inter-node causality, i.e., message Y sent by a component corresponds to message Y' received by another component.

The online monitor is implemented in Xen 3.1.0 [5] running on x86 32-bit architecture. The guest OS is Linux 2.6.18. Xen’s para-virtualization technique modifies the guest OS so that privileged instructions are handled properly by the VMM. Xen uses hypercalls to hand control from guest OS to the VMM when needed. Hypercalls are inserted at various places within the modified guest OS. In Xen’s terminology, a VM is called a *domain*. Xen runs a special domain called *Domain0*, which executes management tasks and performs I/O operations on behalf of other domains.

Below we first describe how vPath’s online monitor tracks thread activities and TCP connections, and then describe the offline log analyzer.

3.1 Monitoring Thread Activities

vPath needs to track which thread performs a `send` or `recv` system call over which TCP connection. If thread scheduling activities are visible to the VMM, it would be easy to identify the running threads. However, unlike process switching, thread context switching is transparent to the VMM. For a process switch, the guest OS has to update the CR3 register to reload the page table base address. This is a privileged operation and generates a trap that is captured by the VMM. By contrast, a thread context switch is not a privileged operation and does not result in a trap. As a result, it is invisible to the VMM.

Luckily, this is not a problem for vPath, because vPath’s task is actually simpler. We only need information about currently active thread when a network send or receive operation occurs (as opposed to fully discovering thread-schedule orders). Each thread has a dedicated stack within its process’s address space. It is unique to the thread throughout its lifetime. This suggests that the VMM could use the stack address in a system call to identify the calling thread. The x86 architecture uses the EBP register for the stack frame base address. Depending on the function call depth, the content of the EBP may vary on each system call, pointing to an address in the thread’s stack. Because the stack has a limited size, only the lower bits of the EBP register vary. Therefore, we can get a stable thread identifier by masking out the lower bits of the EBP register.

Specifically, vPath tracks network-related thread activities as follows:

- The VMM intercepts all system calls that send or receive TCP messages. Relevant system calls in Linux are `read()`, `write()`, `readv()`, `writev()`, `recv()`, `send()`, `recvfrom()`, `sendto()`, `recvmsg()`, `sendmsg()`, and `sendfile()`. Intercepting system calls of a *para-virtualized* Xen VM is possible because they use “`int 80h`” and this software trap can be intercepted by VMM.
- On system call interception, vPath records the current DomainID, the content of the CR3 register, and the content of the EBP register. DomainID identifies a VM. The content of CR3 identifies a process in the given VM. The content of EBP identifies a thread within the given process. vPath uses a combination of DomainID/CR3/EBP to uniquely identify a thread.

By default, system calls in Xen 3.1.0 are not intercepted by the VMM. Xen maintains an IDT (Interrupt Descriptor Table) for each guest OS and the `0x80h` entry corresponds to the system call handler. When a guest OS boots, the `0x80h` entry is filled with the address of the guest OS’s system call handler through the `set_trap_table` hypercall. In order to intercept system calls, we prepare our custom system call handler, register it into IDT, and disable direct registration of the guest OS system call handler. On a system call, vPath checks the type of the system call, and logs the event only if it is a network send or receive operation.

Contrary to the common perception that system call interception is expensive, it actually has negligible impact on performance. This is because system calls already cause a user-to-kernel mode switch. vPath code is only executed after this mode switch and does not incur this cost.

3.2 Monitoring TCP Connections

On a TCP send or receive system call, in addition to identifying the thread that performs the operation, vPath also needs to log the four-element tuple (*source_IP*, *source_port*, *dest_IP*, *dest_port*) that uniquely identifies the TCP connection. This information helps match a send operation in the message source component with the corresponding receive operation in the message destination component. The current vPath prototype adds a hypercall in the guest OS to deliver this information down to the VMM. Upon entering a system call of interest, the modified guest OS maps the socket descriptor number into (*source_IP*, *source_port*, *dest_IP*, *dest_port*), and then invokes the hypercall to inform the VMM.

This approach works well in the current prototype, and it modifies fewer than 100 lines of source code in the guest OS (Linux). However, our end goal is to implement a pure VMM-based solution that does not modify the guest OS at all. Such a pure solution would be easier to deploy in a Cloud Computing platform such as EC2 [2], because it only modifies the VMM, over which the platform service provider has full control.

As part of our future work, we are exploring several techniques to avoid modifying the guest OS. Our early results show that, by observing TCP/IP packet headers in *Domain0*, four-element TCP identifiers can be mapped to socket descriptor numbers observed in system calls with high accuracy. Another alternative technique we are exploring is to have the VMM keep track of the mapping from socket descriptor numbers to four-element TCP identifiers, by monitoring system calls that affect this mapping, including `bind()`, `accept()`, `connect()`, and `close()`.

3.3 Offline Log Analyzer

The offline log analyzer parses logs generated by the online monitor to extract request-processing paths and their performance characteristics. The analyzer’s algorithm is shown in Algorithm 1. The format of input data is shown in Figure 5.

On Line 2 of Algorithm 1, it verifies whether the trace file is in a correct format. On Line 3, it merges the system call log and the hypercall log into a single one for ease of processing. All events are then read into linked lists \mathcal{L} on Line 4.

Events are normalized prior to actual processing. If an application-level message is large, it may take multiple system calls to send the message. Similarly, on the destination, it may take multiple system calls to read in the entire message. These consecutive `send` or `recv` events logically belong to a single operation. On Line 5, multiple consecutive `send` events are merged into a single one. Consecutive `recv` events are merged similarly.

On Line 6, `UPDATERECVTIME` performs another type of event normalization. It updates the timestamp of a

Format of Data Obtained Through System Call Interception						
Event #	Domain #	Time Stamp	CR3	EBP	EAX	EBX
Format of Data Obtained Through Hypercall in Syscall Handler						
Event #	OP Type (R/S)	Domain #	Socket Descriptor #	Local IP Addr & Port	Remote IP Addr & Port	
Example						
0733	Dom1	002780	cr3:04254000	ebp:bfe37034	eax:3	ebx:12
0734	R Dom1	sd:12	L:130.203.8.24:41845	R:130.203.8.25:8009		←
0735	Dom1	002781	cr3:04254000	ebp:bfe34b34	eax:146	ebx:11
0736	S Dom1	sd:11	L:130.203.8.24:80	R:130.203.65.112:2395		←

Figure 5: Format of vPath log data. The example shows two system calls (events 0733 and 0735). For each system call, a hypercall immediately follows (events 0734 and 0736). The IP and port information provided by the hypercall helps identify TCP connections. In the system call log, EAX holds system call number. EBX holds socket descriptor number for read, and write. If EAX is 102 (i.e., `socketcall`), then EBX is the subclass of the system call (e.g. `send` or `recv`).

`recv` event to reflect the end of the receive operation rather than the beginning of the operation. The vPath online monitor records a timestamp for each system call of interest when it is invoked. When a thread sends out a request message and waits for the reply, this event is recorded by vPath and the thread may wait in the blocked state for a long time. To accurately calculate the response time of this remote invocation from the caller side, we need to know when the `recv` operation returns rather than when it starts. For a `recv` system call r performed by a thread T , we simply use the timestamp of the next system call invoked by thread T as the return time of r .

The operation from Line 10 to 17 pairs up a `send` event at the message source with the corresponding `recv` event at the message destination. Once a pair of matching events e_c and e_d are identified, the same TCP connection's events after e_c and e_d are paired up sequentially by `PAIRUPFOLLOWINGS`.

Inside `FINDREMOTEMATCHINGEVENT` on Line 13, it uses a four-element tuple $(source_IP, source_port, dest_IP, dest_port)$ to match a TCP connection tcp_1 observed on a component c_1 with a TCP connection tcp_2 observed on another component c_2 . Suppose c_1 is the client side of the TCP connection. The first `send` operation over tcp_1 observed on c_1 matches with the first `recv` operation over tcp_2 observed on c_2 , and so forth. There is one caveat though. Because port numbers are reused across TCP connections, it is possible that two TCP connections that exist at different times have identical $(source_IP, source_port, dest_IP, dest_port)$. For example, two TCP connections tcp_2 and tcp'_2 that exist on c_2 at different times both can potentially match with tcp_1 on c_1 . We use timestamps to solve this problem. Note that the lifetimes of tcp_2 and tcp'_2 do not overlap and must be far apart, because in modern OS implementa-

Algorithm 1 THE OFFLINE LOG ANALYZER:

Input: Log file \mathcal{F}_i for application process P_i , $1 \leq i \leq n$.
Output: Linked lists \mathcal{L}_i of events, where every event is tagged with the identifier of the user request that triggers the event.

```

1: for each process  $i$  do
2:    $\text{CHECKDATAINTEGRITY}(\mathcal{F}_i)$ 
3:    $\text{PREPROCESSDATA}(\mathcal{F}_i)$ 
4:    $\mathcal{L}_i \leftarrow \text{BUILDEVENTLIST}(\mathcal{F}_i)$ 
5:    $\text{MERGECONSECUTIVEEVENTS}(\mathcal{L}_i)$ 
6:    $\text{UPDATERCVTIME}(\mathcal{L}_i)$ 
7:    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \text{FINDFRONTENDPROCESS}(\mathcal{L}_i)$ 
8: end for
9: /* Pair up every send and recv events. */
10: for each process  $c$  do
11:   for each event  $e_c$  with  $e_c.peer = \text{NULL}$  do
12:      $d \leftarrow \text{FINDPROCESS}(e_c.remote\_IP)$ 
13:      $e_d \leftarrow \text{FINDREMOTEMATCHINGEVENT}(d,$ 
14:        $e_c.local\_IP\&port, e_c.remote\_IP\&port)$ 
15:      $\text{PAIRUPFOLLOWINGS}(e_c, e_d)$ 
16:   end for
17: end for
18: /* Assign a unique ID to each user request. */
19:  $\mathcal{R} \leftarrow \text{IDENTIFYREQUESTS}(\mathcal{Q})$ 
20: for each request id  $r \in \mathcal{R}$  do
21:   while (any event is newly assigned  $r$ ) do
22:     /* Intra-node discovery. */
23:     for each process  $c$  do
24:        $(e_i, e_j) \leftarrow \text{FINDREQUESTBOUNDARY}(c, r)$ 
25:       for all events  $e_k$  within  $(e_i, e_j)$  do
26:         if  $e_k.thread\_id = e_j.thread\_id$  then
27:            $e_k.request\_id \leftarrow r$ 
28:         end if
29:       end for
30:     end for
31:     /* Inter-node discovery. */
32:     for each process  $c$  do
33:        $(e_i, e_j) \leftarrow \text{FINDREQUESTBOUNDARY}(c, r)$ 
34:       for all events  $e_k$  within  $(e_i, e_j)$  do
35:         if  $e_k.request\_id = r$  then
36:            $e_l \leftarrow \text{GETREMOTEMATCHINGEVENT}(e_k)$ 
37:            $e_l.request\_id \leftarrow r$ 
38:         end if
39:       end for
40:     end for
41:   end while
42: end for

```

tions, the ephemeral port used by the client side of a TCP connection is reused only after the entire pool of ephemeral ports have been used, which takes hours or days even for a busy server. This allows a simple solution in vPath. Between tcp_2 and tcp'_2 , we match tcp_1 with the one whose lifetime is closest to tcp_1 . This solu-

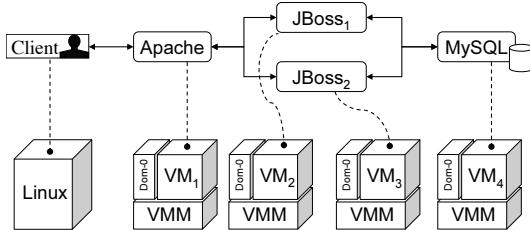


Figure 6: The topology of TPC-W.

tion does not require very accurate clock synchronization between hosts, because the lifetimes of tcp_2 and tcp'_2 are far apart.

On Line 19, all user requests are identified and assigned unique IDs. It goes through events and looks for foreign IP addresses that do not belong to VMs monitored by vPath. Events with foreign IP addresses are generated at front-end components and represent entry/exit points of user requests.

Starting from Line 20, paths are constructed by processing user requests one by one. The algorithm consists of two `for` loops, which implements intra-node discovery and inter-node discovery, respectively. In the first loop, the starting event and ending event of a given request are identified through `FINDREQUESTBOUNDARY`. All events between them and with the same thread ID are assigned the same user request ID. In the second loop (for inter-node discovery), `FINDREQUESTBOUNDARY` is called again to find the starting event and the ending event of every user request. For each event e_k that belongs to the user request, `GETREMOTEMATCHINGEVENT` uses information computed on Line 13 to find the matching event e_l at the other end of the TCP connection. Event e_l is assigned event e_k 's user request ID. This process repeats until every event is assigned a user request ID.

4 Experimental Evaluation

Our experimental testbed consists of Xen VMMs (v3.1.0) hosted on Dell servers connected via Gigabit Ethernet. Each server has dual Xeon 3.4 GHz CPUs with 2 MB of L1 cache and 3 GB RAM. Each of our servers hosts several virtual machines (VMs) with each VM assigned 300 MB of RAM. We use the `xentop` utility in `Domain0` to obtain the CPU utilization of all the VMs running on that server.

4.1 Applications

To demonstrate the generality of vPath, we evaluate vPath using a diverse set of applications written in different programming languages (C, Java, and PHP), developed by communities with very different backgrounds.

TPC-W: To evaluate the applicability of vPath for realistic workloads, we use a three-tier implementation of

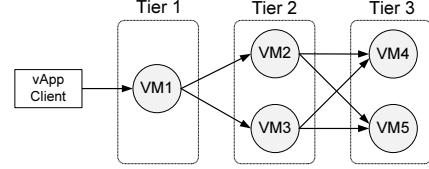


Figure 7: The topology of vApp used in evaluation.

the TPC-W benchmark [27], which represents an online bookstore developed at New York University [31]. Our chosen implementation of TPC-W is a fully J2EE compliant application, following the “Session Facade” design pattern. The front-end is a tier of Apache HTTP servers configured to load balance the client requests among JBoss servers in the middle tier. JBoss 3.2.8SP1 [14] is used in the middle tier. MySQL 4.1 [17] is used for the back-end database tier. The topology of our TPC-W setup is shown in Figure 6. We use the workload generator provided with TPC-W to simulate multiple concurrent clients accessing the application.

This setup is a heterogeneous test environment for vPath. The Apache HTTP server is written in C and is configured to use a multi-process architecture. JBoss is written in Java and MySQL is written in C.

RUBiS: RUBiS [24] is an e-Commerce benchmark developed for academic research. It implements an online auction site loosely modeled after eBay, and adopts a two-tier architecture. A user can register, browse items, sell items, or make a bid. It is available in three different versions: Java Servlets, EJB, and PHP. We use the PHP version of RUBiS in order to differentiate from TPC-W, which is written in Java and also does e-Commerce. Our setup uses one VM to run a PHP-enabled Apache HTTP server and another VM to run MySQL.

MediaWiki: MediaWiki [16] is a mainstream open source application. It is the software behind the popular Wikipedia site (wikipedia.org), which ranks in the top 10 among all Web sites in terms of traffic. As mature software, it has a large set of features, e.g., support for rich media and a flexible namespace. Because it is used to run Wikipedia, one of the highest traffic sites on the Internet, its performance and scalability have been highly optimized. It is interesting to see whether the optimizations violate the assumptions of vPath (i.e., synchronous remote invocation and event causality observable through thread activities) and hence would fail our technique. MediaWiki adopts a two-tier architecture and is written in PHP. Our setup uses one VM to run PHP-enabled Apache and another VM to run MySQL.

vApp: vApp is our own prototype application. It is an extreme test case we designed for vPath. It can exercise vPath with arbitrarily complex request-processing paths. It is a custom multi-tier multi-threaded application written in C. Figure 7 shows an example of a three-tier vApp

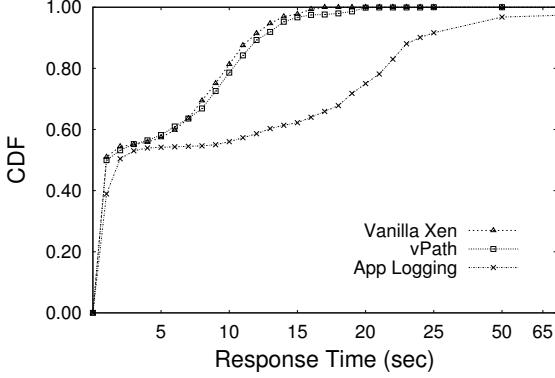


Figure 8: CDF (cumulative distribution function) comparison of TPC-W response time.

Configuration	Response time in seconds (Degradation in %)		Throughput(req/sec) (Degradation in %)
	Average	90 th percentile	
Vanilla Xen	4.45	11.58	4.88
vPath	4.72 (6%)	12.28 (6%)	4.59 (6%)
App Logging	10.31 (132%)	23.95 (107%)	4.10 (16%)

Table 1: Response time and throughput of TPC-W. “App Logging” represents a log-based tracking technique that turns on logging on all tiers of TPC-W.

topology. vApp can form various topologies, with the desired number of tiers and the specified number of servers at each tier. When a server in one tier receives a request, it either returns a reply, or sends another request to one of the servers in the downstream tier. When a server receives a reply from a server in the downstream tier, it either sends another request to a server in the downstream tier, or returns a reply to the upstream tier. All decisions are made based on specified stochastic processes so that it can generate complex request-processing paths with different structures and path lengths.

We also developed a vApp client to send requests to the front tier of the vApp servers. The client can be configured to emulate multiple concurrent sessions. As request messages travel through the components of the vApp server, the identifiers of visited components are appended to the message. When a reply is finally returned to the client, it reads those identifiers to precisely reconstruct the request-processing path, which serves as the ground truth to evaluate vPath. The client also tracks the response time of each request, which is compared with the response time estimated by vPath.

4.2 Overhead of vPath

We first quantify the overhead of vPath, compared with both vanilla (unmodified) Xen and log-based tracking techniques [32, 25]. For the log-based techniques, we turn on logging on all tiers of TPC-W. The experiment below uses the TPC-W topology in Figure 6.

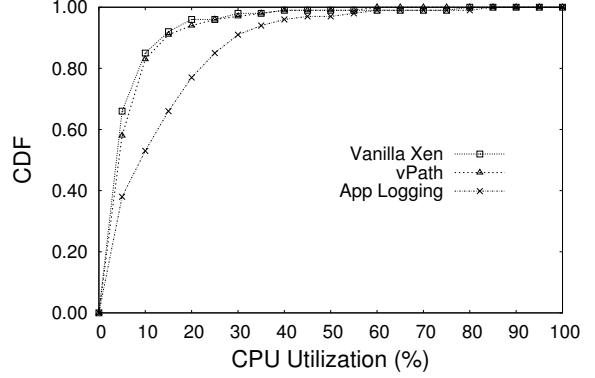


Figure 9: CDF Comparison of TPC-W JBoss tier’s CPU utilization.

Overhead of vPath on TPC-W. Table 1 presents the average and 90th percentile response time of TPC-W benchmark as seen by the client, catering to 100 concurrent user sessions. For all configurations, 100 concurrent sessions cause near 100% CPU utilization at the database tier. Table 1 shows that vPath has low overhead. It affects throughput and average response time by only 6%. By contrast, “App Logging” decreases throughput by 16% and increases the average response time by as high as 132%. The difference in response time is more clearly shown in Figure 8, where vPath closely follows “vanilla Xen”, whereas “App Logging” significantly trails behind.

Figure 9 shows the CPU utilization of the JBoss tier when the database tier is saturated. vPath has negligible CPU overhead whereas “App Logging” has significant CPU overhead. For instance, vPath and “vanilla Xen” have almost identical 90th percentile CPU utilization (13.6% vs. 14.4%), whereas the 90th percentile CPU utilization of “App Logging” is 29.2%, more than twice that of vPath. Thus, our technique, by eliminating the need for using application logging to trace request-processing paths, improves application performance and reduces CPU utilization (and hence power consumption) for data centers. Moreover, vPath eliminates the need to repeatedly write custom log parsers for new applications. Finally, vPath can even work with applications that cannot be handled by log-based discovery methods because those applications were not developed with this requirement in mind and do not generate sufficient logs.

Overhead of vPath on RUBiS. Due to space limitation, we report only summary results on RUBiS. Table 2 shows the performance impact of vPath on RUBiS. We use the client emulator of RUBiS to generate workload. We set the number of concurrent user sessions to 900 and set user think time to 20% of the original value in order to drive the CPU of the Apache tier (which runs PHP) to 100% utilization. vPath imposes low overhead on RUBiS, decreasing throughput by only 5.6%.

	Response Time in millisec (Degradation in %)	Throughput in req/sec (Degradation in %)
Vanilla Xen	597.2	628.6
vPath	681.8 (14.13%)	593.4 (5.60%)

Table 2: Performance impact of vPath on RUBiS.

Configuration	Response time (in sec)		Throughput (req/sec)	
	Avg(Std.)	Overhead	Avg(Std.)	Overhead
Vanilla Xen	1.69(.053)		2915.1(88.9)	
(1) Intercept Syscall	1.70(.063)	.7%	2866.6(116.5)	1.7%
(2) Hypercall	1.75(.050)	3.3%	2785.2(104.6)	4.5%
(3) Transfer Log	2.02(.056)	19.3%	2432.0(58.9)	16.6%
(4) Disk Write	2.10(.060)	23.9%	2345.4(62.3)	19.1%

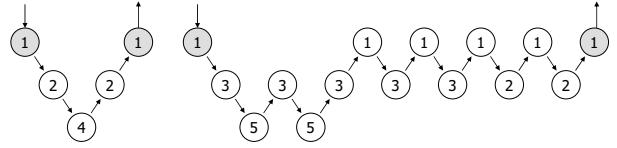
Table 3: Worst-case overhead of vPath and breakdown of the overhead. Each row represents the overhead of the previous row plus the overhead of the additional operation on that row.

Worst-case Overhead of vPath. The relative overhead of vPath depends on the application. We are interested in knowing the *worst-case* overhead (even if the worst case is unrealistic for practical systems).

The relative overhead of vPath can be calculated as $\frac{v}{v+p}$, where v is vPath's processing time for monitoring a network send or receive operation, and p is the application's processing time related to this network operation, e.g., converting data retrieved from the database into HTML and passing the data down the OS kernel's network stack. vPath's relative overhead is highest for an application that has the lowest processing time p . We use a tiny echo program to represent such a worst-case application, in which the client sends a one-byte message to the server and the server echoes the message back without any processing. In our experiment, the client creates 50 threads to repeatedly send and receive one-byte messages in a busy loop, which fully saturates the server's CPU.

When the application invokes a network send or receive system call, vPath performs a series of operations, each of which introduces some overhead: **(1)** intercepting system call in VMM, **(2)** using hypercall to deliver TCP information (*src_IP*, *src_port*, *dest_IP*, *dest_port*) from guest OS to VMM, **(3)** transferring log data from VMM to *Domain0*, and **(4)** *Domain0* writing log data to disk. These operations correspond to different rows in Table 3, where each row represents the overhead of the previous row plus the overhead of the additional operation on that row.

Table 3 shows that intercepting system calls actually has negligible overhead (1.7% for throughput). The biggest overhead is due to transferring log data from VMM to *Domain0*. This step alone degrades throughput by 12.1%. Our current implementation uses VMM's `printk()` to transfer log data to *Domain0*, and we are exploring a more efficient implementation. Combined



(a) Simple path

(b) Complex path

Figure 10: Examples of vApp’s request-processing paths discovered by vPath. The circled numbers correspond to VM IDs in Figure 7.

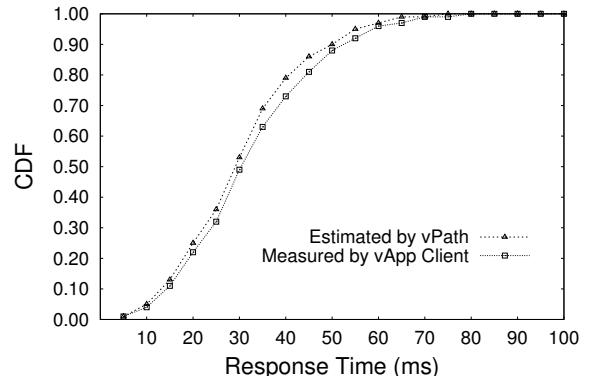


Figure 11: CDF of vApp’s response time, as estimated by vPath and actually measured by the vApp client.

together, the operations of vPath degrade throughput by 19.1%. This is the worst-case for a contrived tiny “application.” For real applications, throughput degradation is much lower, only 6% for TPC-W and 5.6% for RUBiS.

4.3 Request-Processing Paths of vApp

Our custom application vApp is a test case designed to exercise vPath with arbitrarily complex request-processing paths. We configure vApp to use the topology in Figure 7. The client emulates 10-30 concurrent user sessions. In our implementation, as a request message travels through the vApp servers, it records the actual request-processing path, which serves as the ground truth to evaluate vPath.

The request-processing path of vApp, as described in 4.1, is designed to be random. To illustrate the ability of our technique to discover sophisticated request-processing paths, we present two discovered paths in Figure 10. The simple path consists of 2 remote invocations in a linear structure, while the complex path consists of 7 invocations and visits some components more than once.

In addition to discovering request-processing paths, vPath can also accurately calculate the end-to-end response times as well as the time spent on each tier along a path. This information is helpful in debugging distributed systems, e.g., identifying performance bottlenecks and abnormal requests. Figure 11 compares the

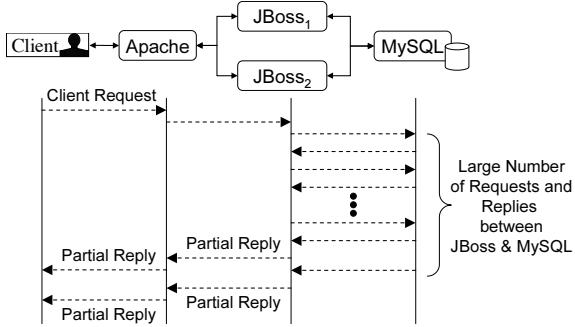


Figure 12: Typical request-processing paths of TPC-W.

end-to-end response time estimated by vPath with that actually measured by the vApp client. The response time estimated by vPath is almost identical to that observed by the client, but slightly lower. This small difference is due to message delay between the client and the first tier of vApp, which is not tracked by vPath because the client runs on a server that is not monitored by vPath.

We executed a large number of requests at different session concurrency levels. We also experimented with topologies much larger than that in Figure 7, with more tiers and more servers in each tier. All the results show that vPath precisely discovers the path of each and every executed request.

4.4 Request-Processing Paths of TPC-W

The three-tier topology (see the top of Figure 12) of the TPC-W testbed is static, but its request-processing paths are dynamic and can vary, depending on which JBoss server is accessed and how many queries are exchanged between JBoss and MySQL. The TPC-W client generates logs that include the total number of requests, current session counts, and individual response time of each request, which serve as the ground truth for evaluating vPath. In addition to automated tests, for the purpose of careful validation, we also conduct eye-examination on some samples of complex request-processing paths discovered by vPath and compare them with information in the application logs.

vPath is able to correctly discover all request-processing paths with 100% completeness and 100% accuracy (see Section 2.1 for the definition). We started out without knowing how the paths of TPC-W would look. From the results, we were able to quickly learn the path structure without any knowledge of the internals of TPC-W. Typical request-processing paths of TPC-W have the structure in Figure 12.

We observe two interesting things that we did not anticipate. First, when processing one request, JBoss makes a large number of invocations to MySQL. Most requests fall into one of two types. One type makes about 20 invocations to MySQL, while the other type makes

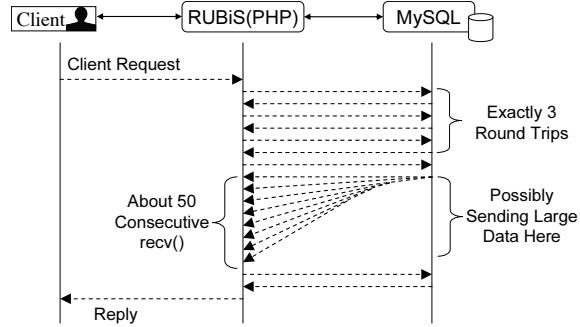


Figure 13: Typical request-processing paths of RUBiS.

about 200 invocations. These two types represent radically different TPC-W requests.

The second interesting observation with TPC-W is that, both JBoss and Apache send out replies in a pipeline fashion (see Figure 12). For example, after making the last invocation to MySQL, JBoss reads in partial reply from MySQL and immediately sends it to Apache. JBoss then reads and sends the next batch of replies, and so forth. This pipeline model is an effort to reduce memory buffer, avoid memory copy, and reduce user-perceived response time. In this experiment, once JBoss sends the first partial reply to Apache, it no longer makes invocations to MySQL (it only reads more partial replies from MySQL for the previous invocation). vPath is general enough to handle an even more complicated case, where JBoss sends the first partial reply to Apache, and then makes more invocations to MySQL in order to retrieve data for constructing more replies. Even for this complicated, hypothetical case, all the activities will still be correctly assigned to a single request-processing path.

4.5 Request-Processing Paths of RUBiS and MediaWiki

Unlike TPC-W, which is a benchmark intentionally designed to exercise a breadth of system components associated with e-Commerce environments, RUBiS and MediaWiki are designed with practicality in mind, and their request-processing paths are actually shorter and simpler than those of TPC-W.

Figure 13 shows the typical path structure of RUBiS. With vPath, we are able to make some interesting observations without knowing the implementation details of RUBiS. We observe that a client request first triggers three rounds of messages exchanged between Apache and MySQL, followed by the fourth round in which Apache retrieves a large amount of data from MySQL. The path ends with a final round of messages exchanged between Apache and MySQL. The pipeline-style partial message delivery in TPC-W is not observed in RUBiS. RUBiS and TPC-W also differ significantly in their database access patterns. In TPC-W, JBoss makes many

small database queries, whereas in RUBiS, Apache retrieves a large amount of data from MySQL in a single step (the fourth round). Another important difference is that, in RUBiS, many client requests finish at Apache without triggering database accesses. These short requests are about eight times more frequent than the long ones. Finally, in RUBiS, Apache and MySQL make many DNS queries, which are not observed in TPC-W.

For MediaWiki, the results of vPath show that very few requests actually reach all the way to MySQL, while most requests are directly returned by Apache. This is because there are many static contents, and even for dynamic contents, MediaWiki is heavily optimized for effective caching. For a typical request that changes a wiki page, the PHP module in Apache makes eight accesses to MySQL before replying to the client.

4.6 Discussion on Benchmark Applications

We started the experiments with little knowledge of the internals of TPC-W, RUBiS and MediaWiki. During the experimentation, we did not read their manuals or source code. We did not modify their source code, bytecode, or executable binary. We did not try to understand their application logs or write parsers for them. We did not install any additional application monitoring tools such as IBM Tivoli or HP OpenView. In short, we did not change anything in the user space.

Yet, with vPath, we were able to make many interesting observations about the applications. Especially, different behaviors of the applications made us wonder, in general how to select “representative” applications to evaluate systems performance research. TPC-W is a widely recognized *de facto* e-Commerce benchmark, but its behavior differs radically from the more practical RUBiS and MediaWiki. This discrepancy could result from the difference in application domain, but it is not clear whether the magnitude of the difference is justified. We leave it as an open question rather than a conclusion.

This question is not specific to TPC-W. For example, the Trade6 benchmark [35] developed by IBM models an online stock brokerage Web site. We have intimate knowledge of this application. As both a benchmark and a testing tool, it is intentionally developed with certain complexity in mind in order to fully exercise the rich functions of WebSphere Application Server. It would be interesting to know, to what degree the conclusions in systems performance research are misguided by the intentional complexity in benchmarks such as TPC-W and Trade6.

5 Related Work

There is a large body of work related to request-processing path discovery. They can be broadly classified into two categories: statistical inference and system-dependent instrumentation. The statistical approach

takes readily available information (e.g., the arrival time of network packets) as inputs, and infers request-processing paths in a “most likely” way. Its accuracy degrades as the workload increases, because activities of concurrent requests are mingled together and hard to differentiate. The instrumentation approach may accurately discover request-processing paths, but its applicability is limited due to its intrusive nature. It requires knowledge (and often source code) of the specific middleware or applications in order to do instrumentation.

5.1 Statistical Inference

Aguilera et al. [1] proposed two algorithms for debugging distributed systems. The first algorithm finds nested RPC calls and uses a set of heuristics to infer the causality between nested RPC calls, e.g., by considering time difference between RPC calls and the number of potential parent RPC calls for a given child RPC call. The second algorithm only infers the average response time of components; it does not build request-processing paths.

WAP5 [21] intercepts network related system calls by dynamically re-linking the application with a customized system library. It statistically infers the causality between messages based on their timestamps. By contrast, our method is intended to be precise. It monitors thread activities in order to accurately infer event causality.

Anandkumar et al. [3] assumes that a request visits distributed components according to a known semi-Markov process model. It infers the execution paths of individual requests by probabilistically matching them to the footprints (e.g., timestamped request messages) using the maximum likelihood criterion. It requires synchronized clocks across distributed components. Spaghetti is evaluated through simulation on simple hypothetical process models, and its applicability to complex real systems remains an open question.

Sengupta et al. [25] proposed a method that takes application logs and a prior model of requests as inputs. However, manually building a request-processing model is non-trivial and in some cases prohibitive. In some sense, the request-processing model is in fact the information that we want to acquire through monitoring. Moreover, there are difficulties with using application logs as such logs may not follow any specific format and, in many cases, there may not even be any logs available.

Our previous work [32] takes an unsupervised learning approach to infer attributes (e.g., thread ID, time, and Web service endpoint) in application logs that can link activities observed on individual servers into end-to-end paths. It requires synchronized clocks across distributed components, and the discovered paths are only statistically accurate.

5.2 System-dependent Instrumentation

Magpie [4] is a tool-chain that analyzes event logs to infer a request’s processing path and resource consumption. It can be applied to different applications but its inputs are application dependent. The user needs to modify middleware, application, and monitoring tools in order to generate the needed event logs. Moreover, the user needs to understand the syntax and semantics of the event logs in order to manually write an event schema that guides Magpie to piece together events of the same request. Magpie does kernel-level monitoring for measuring resource consumption, but not for discovering request-processing paths.

Pip [20] detects problems in a distributed system by finding discrepancies between actual behavior and expected behavior. A user of Pip adds annotations to application source code to log messaging events, which are used to reconstruct request-processing paths. The user also writes rules to specify the expected behaviors of the requests. Pip then automatically checks whether the application violates the expected behavior.

Pinpoint [9] modifies middleware to inject end-to-end request IDs to track requests. It uses clustering and statistical techniques to correlate the failures of requests to the components that caused the failures.

Chen et al. [8] used request-processing paths as the key abstraction to detect and diagnose failures, and to understand the evolution of a large system. They studied three examples: Pinpoint, ObsLogs, and SuperCall. All of them do intrusive instrumentation in order to discover request-processing paths.

Stardust [30] uses source code instrumentation to log application activities. An end-to-end request ID helps recover request-processing paths. Stardust stores event logs into a database, and uses SQL statements to analyze the behavior of the application.

5.3 Inferring Dependency from System Call

BackTracker [15] is a tool that helps find the source event of an intrusion, backtracking from the point when the intrusion is detected. It logs system calls to help infer dependency between system resources, but does not monitor thread activities and network operations.

Taser [12] is a system that helps recover files damaged by an intrusion. Like BackTracker, it also uses information logged from system calls to infer the dependency of system resources. It monitors network operations, but does not monitor thread activities and does not attempt to precisely infer message causality. Moreover, both BackTracker and Taser are designed for a single server. They do not track dependency across servers.

Kai et al. [26] proposed a method that uses an optional field of TCP packets to track inter-node causality, and assumes that intra-node causality is only introduced by process/thread forking. As a result, this method cannot

handle the case where intra-node causality is caused by thread synchronization, e.g., a dispatcher thread wakes up a worker thread to process an incoming request. This is a wide used programming pattern in thread pooling.

6 Concluding Remarks

We studied the important problem of finding end-to-end request-processing paths in distributed systems. We proposed a method, called *vPath*, that can precisely discover request-processing paths for most of the existing mainstream software. Our key observation is that the problem of request-processing path discovery can be radically simplified by exploiting programming patterns widely adopted in mainstream software: (1) synchronous remote invocation, and (2) assigning a thread to do most of the processing for an incoming request.

Following these observations to infer event causality, our method can discover request-processing paths from minimal information recorded at runtime—which thread performs a send or receive system call over which TCP connection. This information can be obtained efficiently in either OS kernel or VMM without modifying any user-space code.

We demonstrated the generality of *vPath* by evaluating with a diverse set of applications (TPC-W, RUBiS, MediaWiki, and the home-grown *vApp*) written in different programming languages (C, Java, and PHP). *vPath*’s online monitor is lightweight. We found that activating *vPath* affects the throughput and average response time of TPC-W by only 6%

Acknowledgments

Part of this work was done during Byung Chul Tak’s summer internship at IBM. We thank IBM’s Yaoping Ruan for helpful discussions and Fausto Bernardini for the management support. We thank the anonymous reviewers and our shepherd Landon Cox for their valuable feedback. The PSU authors were funded in part by NSF grants CCF-0811670, CNS-0720456, and a gift from Cisco, Inc.

References

- [1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP’03: Proceedings of the 19th Symposium on Operating Systems Principles*, pages 74–89, New York, NY, USA, 2003. ACM.
- [2] Amazon Elastic Compute Cloud. <http://aws.amazon.com/ec2/>.
- [3] A. Anandkumar, C. Bisdikian, and D. Agrawal. Tracking in a spaghetti bowl: monitoring transactions using footprints. In *SIGMETRICS ’08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 133–144, New York, NY, USA, 2008. ACM.
- [4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling.

- In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, Berkeley, CA, USA, 2004. USENIX Association.
- [5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)*, 2003.
 - [6] R. V. Behren, J. Condit, and E. Brewer. Why Events Are A Bad Idea (for high-concurrency servers). In *Proceedings of HotOS IX*, 2003.
 - [7] R. V. Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *In Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM Press, 2003.
 - [8] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI'04: Proceedings of the 1st conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2004. USENIX Association.
 - [9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.
 - [10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI'05: Proceedings of the 2nd conference on Networked Systems Design & Implementation*, 2005.
 - [11] T. Erl. *Service-oriented architecture*. Prentice Hall, 2004.
 - [12] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *SOSP '05: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 163–176, New York, NY, USA, 2005. ACM.
 - [13] IBM SOA Infrastructure Consulting Services. http://www-935.ibm.com/services/us/its/pdf/br_infrastructure-architecture-healthcheck-for-soa.pdf.
 - [14] The JBoss Application Server. <http://www.jboss.org>.
 - [15] S. T. King and P. M. Chen. Backtracking Intrusions. In *SOSP'03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 74–89, New York, NY, USA, 2003. ACM.
 - [16] MediaWiki. <http://www.mediawiki.org>.
 - [17] MySQL. <http://www.mysql.com>.
 - [18] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: an efficient and portable web server. In *ATEC '99: Proceedings of USENIX Annual Technical Conference*, Berkeley, CA, USA, 1999. USENIX Association.
 - [19] W. D. Pauw, R. Hoch, and Y. Huang. Discovering conversations in web services using semantic correlation analysis. volume 0, pages 639–646, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
 - [20] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, Berkeley, CA, USA, 2006. USENIX Association.
 - [21] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 347–356, New York, NY, USA, 2006. ACM.
 - [22] Y. Ruan and V. Pai. Making the “box” transparent: system call performance as a first-class result. In *Proceedings of the USENIX Annual Technical Conference 2004*. USENIX Association Berkeley, CA, USA, 2004.
 - [23] Y. Ruan and V. Pai. Understanding and Addressing Blocking-Induced Network Server Latency. In *Proceedings of the USENIX Annual Technical Conference 2006*. USENIX Association Berkeley, CA, USA, 2006.
 - [24] RUBIS. <http://rubis.objectweb.org/>.
 - [25] B. Sengupta and N. Banerjee. Tracking transaction footprints for non-intrusive end-to-end monitoring. *Autonomic Computing, International Conference on*, 0:109–118, 2008.
 - [26] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 189–200, New York, NY, USA, 2008. ACM.
 - [27] W. Smith. TPC-W: Benchmarking An Ecommerce Solution. <http://www.tpc.org/information/other/techarticles.asp>.
 - [28] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proceedings of the 2nd Symposium on NSDI'05, Boston MA*, May 2005.
 - [29] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A Scalable Application Placement Algorithm for Enterprise Data Centers. In *WWW*, 2007.
 - [30] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, New York, NY, USA, 2006. ACM.
 - [31] NYU TPC-W. <http://www.cs.nyu.edu/pdsg/>.
 - [32] T. Wang, C. shing Perng, T. Tao, C. Tang, E. So, C. Zhang, R. Chang, and L. Liu. A temporal data-mining approach for discovering end-to-end transaction flows. In *2008 IEEE International Conference on Web Services (ICWS08)*, Beijing, China, 2008.
 - [33] M. Welsh. A Note on the status of SEDA. <http://www.eecs.harvard.edu/~mdw/proj/seda/>.
 - [34] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.
 - [35] H. Yu, J. Moreira, P. Dube, I. Chung, and L. Zhang. Performance Studies of a WebSphere Application, Trade, in Scale-out and Scale-up Environments. In *Third International Workshop on System Management Techniques, Processes, and Services (SMTPS), IPDPS*, 2007.

The Restoration of Early UNIX Artifacts

Warren Toomey

wtoomey@staff.bond.edu.au

School of IT, Bond University

Abstract

UNIX turns 40 this year: many happy returns! Four decades is a vast period for the computing industry: systems from the 1970s now seem rudimentary and primitive. And yet, the early versions of UNIX were epitomes of sophisticated concepts packaged into elegant systems. UNIX' influence has been so powerful that it reverberates down to affect us in the 21st century.

The history of the development of UNIX has been well documented, and over the past decade or so, efforts have been made to find and conserve the software and documentation artifacts from the earliest period of UNIX history. This paper details the work that has been done to restore the artifacts from this time to working order and the lessons learned from this work.

1 Introduction

In 2009, the UNIX¹ operating system celebrates the 40th anniversary of its creation. In the middle of 1969, after AT&T's withdrawal from the Multics project, a number of researchers at AT&T's Bell Labs began the design and development of a simpler operating system which would be named "UNIX" [10]. Led primarily by Ken Thompson and Dennis Ritchie, but with many other colleagues involved, Bell Labs' UNIX would combine several leading-edge concepts (multitasking, a process model, a hierarchical filesystem) and some new concepts (I/O redirection, pipes, portable implementation in a high-level language) to become an elegant and sophisticated system. The 7th Edition of UNIX released in 1979 (and its 32-bit port called "32V") would become the ancestors to all of the hundreds of UNIX-derived systems that now exist² including AIX, Solaris, Apple's Darwin kernel and the various open-source BSD systems. UNIX and the C language would exert a significant influence on the computing industry in the 1980s and 1990s, and see the creation of such vendor-neutral standards as IEEE 1003 POSIX, ISO/IEC 9945, ANSI C and ISO/IEC 9899.

While the history of UNIX has been well-documented [5, 7, 8, 10], there was a time when

the actual artifacts of early UNIX development were in great danger of being lost forever. This has been rectified in the last decade with the collection of a significantly large number of old UNIX systems. Software, however, is simply a collection of zeroes and ones if it is not able to run, and a lot of work has been done to bring these old UNIX systems back to life.

The restoration of a software artifact to working order brings with it a wealth of difficulties: documentation is missing or incomplete, source code is missing leaving only the binary executables, or conversely the source exists but the compilation tools to reconstruct the executables are missing. The restoration of an operating system to working order presents its own issues, as the system needs a suitable hardware environment in which to run, a suitable filesystem and a set of system executables to initialise the system and make it useful.

This paper presents four case studies in software restoration: two early UNIX kernels, the earliest extant C compiler, and a set of executables and source code fragments from 1972. The case studies highlight the above issues in restoration, and outline the approaches taken to resolve the issues.

2 TUHS and the UNIX Archive

In 1995 the UNIX Heritage Society (TUHS)³ was founded with a charter to preserve, maintain and restore historical and non-mainstream UNIX systems. TUHS has been successful in unearthing artifacts from many important historical UNIX systems; this includes system & application source code, system & application executables, user manuals & documentation, and images of populated filesystems.

The proliferation of UNIX variants and the longevity of minicomputer systems such as the VAX and the PDP-11 made TUHS' task of collecting old UNIX systems and their documentation relatively straightforward. Quite quickly the society had gathered such early system as 6th and 7th Edition UNIX, 32V, System III, the BSDs, and some early commercial variants such as Ultrix-11.

The building of an archive of early UNIX systems was initially quite dubious, legally. Most of TUHS' mem-

bers were covered by various UNIX source code licenses from AT&T or USL, but not every license covered the sum of material stored in the archive. TUHS began a process of lobbying SCO,⁴ then owners of the UNIX source code, for some license which would allow access to the material in the archive. With the immense assistance of Dion Johnson at SCO, in 1998 a cheap hobbyist license was made available which gave source-code access to the various PDP-11 UNIX systems, 32V and System III [11]. And in 2002, after much lobbying from TUHS, the PDP-11 UNIX systems and 32V were placed under an open-source BSD-style license.

System	Released	Features
1st Edition	Nov 1971	multitasking, multiuser, hierarchical filesystem
2nd Edition	June 1972	support for memory management on the PDP-11/45
3rd Edition	Feb 1973	pipes and C
4th Edition	Nov 1973	rewritten in the C language
5th Edition	June 1974	first version made available outside Bell Labs
6th Edition	May 1975	ported to multiple platforms
7th Edition	Jan 1979	large filesystem support, the stdio library, many new commands
32V	May 1979	port of 7th Edition to the 32-bit VAX platform

Table of Early UNIX Releases

For a while, it seemed that the archaeology of UNIX stopped somewhere around 1974. The source code and binaries for 5th Edition UNIX existed, but not the files for the manuals; conversely, only the 4th Edition UNIX manuals existed, but not the source code nor any binaries for the system. At the time, Dennis Ritchie told us that there was very little material from before 4th Edition, just some snippets of code listings. Then, around the mid-90s, Paul Vixie and Keith Bostic “unearthed a DECtape drive and made it work”, and were able to read a number of DECtapes which had been found “under the floor of the computer room” at Bell Labs. These tapes would turn out to contain a bounty of early UNIX artifacts.

Two issues immediately arose with the extraction of the tapes’ contents: what format were the tapes in, and the interpretation of the timestamps on the files therein. 7th Edition introduced the *tar(1)* tape archive format; before *tar(1)* there was *rkd(1)* used in 1st Edition to dump an RK05 disk’s raw filesystem to nine DECtapes, *tap(1)* used from 1st to 4th Edition to dump selected parts of a filesystem to DECtape, and *tp(1)* used from 4th to 6th Edition to dump selected parts of a filesystem to tape.

Fortunately, the formats for *tap(1)* and *tp(1)* are documented, and it was simple to write programs to extract the files from both archive formats.

Timestamp interpretation is a much more difficult issue to solve, as Dennis Ritchie noted in a private e-mail:

The difficulty of [timestamp] interpretation [is due] to epoch uncertainty. Earliest Unix used a 32-bit representation of time measured in 60ths of one second, which implies a period of just over 2 years if the number is taken as unsigned. In consequence, during 1969-73, the epoch was changed several times, usually by back-dating existing files on disk and tape and changing the origin.

For each DECtape unearthed and read, the epoch used can only be determined by looking through the contents of the tape and determining where the files should be placed in the known history of UNIX development. We will consider four of the artifacts unearthed in reverse chronological order.

3 The Nsys Kernel: 1973

One of the DECtapes was labelled ‘nsys’, and Dennis Ritchie’s initial e-mail on the tape’s contents noted:

So far as I can determine, this is the earliest version of Unix that currently exists in machine-readable form. ... What is here is just the source of the OS itself, written in the pre-K&R dialect of C. ... It is intended only for PDP-11/45, and has setup and memory-handling code that will not work on other models).

I’m not sure how much work it would take to get this system to boot. Even compiling it might be a bit of a challenge. ... Best wishes with this. I’d be interested to hear whether anyone gets the [system] to run.

Initial interpretation of the timestamps in the archive led us to believe that the files were dated January 1973, but after analysing the contents and their placement in the history of UNIX, we now believe that the files are dated August 1973, just before the release of the 4th Edition of UNIX in November 1973.

Ritchie’s innocuous comments on “how much work it would take to get this system to boot” seemed to be an implicit challenge, and I began the restoration task soon after receiving the tape’s contents. My tools were a working 5th Edition UNIX compiler and environment running on top of the SIMH PDP-11 simulator [2], along with my own Apout emulator (see below).

Restoration work of this kind generally involves consulting existing documentation (in this case, the 4th Edition Programmers Manual and John Lions' Commentary on 6th Edition UNIX [6]), interpreting the few available source code comments,⁵ single-stepping the machine code in the simulator, and intuiting what corrections need to be made based on the system's behaviour.

As predicted by Ritchie, the compilation was a bit of a challenge due to the changes in the C language between 1973 and 1974: sub-structures defined within a structure were delimited by parentheses in ‘nsys’, but by curly braces in 5th Edition. However, the main issue was an incompatibility of the filesystem layout between ‘nsys’ and 5th Edition: the *filsys* structure in 5th Edition has an extra field, *s_ronly*, and the *inode* structure in 5th Edition also has an extra field, *i_1astr*.

One last stumbling block was found which prevented the ‘nsys’ kernel from booting via the 5th Edition’s bootstrap code. While the 5th Edition kernel starts execution at location 0, the ‘nsys’ kernel starts execution at location 2. With a small amount of code transposition, the ‘nsys’ kernel was able to boot on top of a 5th Edition filesystem and behave normally.

There is one last comment to make about the ‘nsys’ kernel. Although the 4th Edition of UNIX (dated November 1973) has the *pipe(2)* system call, and an internal Bell Labs meeting in January 1973⁶ notes the existence of pipes, the ‘nsys’ kernel has *pipe(2)* listed but not implemented in the system call table. 3rd Edition UNIX was the last version written in PDP-11 assembly language. During 1973, while the assembly version was still being developed, the system was also rewritten in the new C language. After discussions with Ritchie, it seems most likely that pipes were implemented in the assembly version of UNIX first, and added to the C version after most of the core functionality had been reimplemented.

4 1st and 2nd Edition Binaries: 1972

Two of the DECtapes read by Bostic, Vixie and Ritchie were labelled ‘s1’ and ‘s2’. Ritchie’s initial notes were:

s1: I haven’t cracked this yet.

s2 (tap format): This is not source, but a dump of (parts of) /bin, /etc, /usr/lib, and bits of a few other directories.

The contents of the ‘s2’ tape, being in *tap(1)* format with timestamps in 60ths of a second, were easy enough to extract but not to date. Most of the files were executables in early UNIX ‘a.out’ format with a mixture of 0405 and 0407 signatures.⁷ This, along with the names and contents of the executables, indicate that the tape was

written at a time around the 2nd Edition of UNIX: files are dated from January 1972 through to February 1973.

Having a set of early UNIX executables is nice, but having them execute is much nicer. There were already a number of PDP-11 emulators available to run executables, but there was a significant catch: with no 1st or 2nd Edition UNIX kernel, the executables would run up to their first system call, and then “fall off the edge of the world” and crash.

Fortunately, there was a solution. As part of my overall early UNIX restoration work, I had written a user-level emulator for UNIX a.out binaries called ‘Apout’.⁸ Like the Wine emulator for Windows, Apout simulates user-mode PDP-11 instructions, but system calls invoked by the TRAP instruction are caught by Apout and emulated by calling equivalent native-mode system calls.

Apout had already been written to run a.out executables from 5th, 6th and 7th Edition UNIX, 2.9BSD and 2.11BSD. Dennis Ritchie had luckily scanned in his paper copy of the 1st Edition Programmers Manual, and I obtained a paper copy of the 2nd Edition Programmers Manual from Norman Wilson. With these in hand, the work to add 1st and 2nd Edition executable support was possible, but not trivial. The PDP-11/20 used by 1st Edition UNIX required an add-on module known as the KE11A Extended Arithmetic Element to perform operations such as multiply or divide. The KE11A needed to be emulated, and I was able to borrow some code written for SIMH to use in Apout. There were other issues to solve, not the least being discrepancies between the UNIX Programmers Manual and the expected behaviour of the system calls being used by the executables (for example, seeks on ordinary files were in bytes, but seeks on device files were in 512-byte blocks). Eventually, a faithful emulation of the 1st and 2nd Edition UNIX executing environment was made, allowing executables such as the early shell, ls, cp, mv, rm and friends to run again:

```
# chdir /
# ls -l
total 32
236 sdrwr- 1024 May 23 14:24:12 bin
568 sdrwr- 512 May 18 06:40:28 dev
297 sdrwr- 512 May 16 03:07:56 etc
299 sdrwr- 512 May 19 07:33:00 tmp
301 sdrwr- 512 May 5 23:10:38 usr
# chdir /bin
# ls -l
total 215
374 sxr-r- 2310 Jan 25 17:20:48 ar
375 lxr-r- 7582 Jun 29 17:45:20 as
377 sxr-r- 2860 Mar 6 12:23:38 cal
378 sxr-r- 134 Jan 16 17:53:34 cat
385 sxr-r- 160 Jan 16 17:53:36 cp
. . .
```

For those unfamiliar with the output from 1st Edition UNIX *ls(1)*, the first column shows the file’s i-node num-

ber. The *s/l* character in the next column indicates if the file is ‘small’ or ‘large’ (4096 bytes or more), the *d/x* indicates if the entry is a directory or executable (there being only one executable bit per file), and the two *rw* entries show the file’s read/write status for owner and other (there being no groups yet).

The ‘s1’ DECTape (noted by Ritchie as “not cracked yet”) proved to be much more intriguing and at the same time extremely frustrating. A block-level analysis showed source code in both C and PDP-11 assembly, none of which appeared to be for the UNIX kernel. There was no apparent archive structure, nor any i-nodes. All of the DECTape appeared to be used, and this led me to conclude that ‘s1’ was one of the middle DECTapes in the set of nine used when *rkd(1)* dumped an RK05 disk’s contents block-by-block out to tape. With the first tape containing the disk’s i-nodes missing, the ‘s1’ tape was merely a collection of 512-byte blocks.

In places, some source files were stored in contiguous blocks, and the few comments inside allowed me to recover the source for such early programs as *ls(1)*, *cat(1)* and *cp(1)*. But for the most part, the arbitrary placement of blocks and lack of comments stymied further file recovery. Setting things aside for a while, I worked on other projects including a tool to compare multiple code trees in C and other languages.⁹ It took nearly two years to realise that I could use this tool to match the fragments from the ‘s1’ tape to source files in other early UNIX systems such as the 5th Edition. Independently and concurrently, Doug Merritt also worked on identifying the source fragments from the ‘s1’ tape, and we used each other’s work to cross-compare and validate the results. In the end, the ‘s1’ tape contained source code for the assembler *as*, the Basic interpreter *bas*, the debugger *db*, the form letter generator *form*, the linker *ld*, and system utilities such as *ar*, *cat*, *chmod*, *chown*, *cmp*, *cp*, *date*, *df*, *getty*, *glob*, *goto*, *if*, *init*, *login* and *ls*.

5 Early C Compilers: 1972

Two other DECTapes recovered by Ritchie contain source code for two of the earliest versions of the original C compiler:¹⁰

The first [tape] is labeled ‘last1120c’, the second ‘prestruct-c’. The first is a saved copy of the compiler preserved just as we were abandoning the PDP-11/20; the second is a copy of the compiler just before I started changing it to use structures itself. ...

The earlier compiler does not know about structures at all: the string “struct” does not appear anywhere. The [later] compiler does implement structures in a way that begins to

approach their current meaning. ... Aside from their small size, perhaps the most striking thing about these programs is their primitive construction, particularly the many constants strewn throughout.

With a lot of handwork, there is probably enough material to construct a working version of the last1120c compiler, where “works” means “turns source into PDP-11 assembler”.

Interpreting the timestamps on the tapes gives a date of July 1972 for the ‘last1120c’ compiler and a date of December 1972 for the ‘prestruct-c’ compiler. Again, Ritchie’s note that “there is probably enough material to construct a working version of the last1120c compiler” was taken as an implicit challenge to bring these compilers back to life. But there was a “chicken and egg” problem here: both compilers are in such a primitive dialect of C that no extant working compilers would be able to parse their source code. Good fortune was, however, on my side. Not only did the ‘s2’ tape contain early UNIX system executables, but hidden away in /usr/lib were executables named *c0* and *c1*: the two passes of an early C compiler. It seemed likely that these executables running on the Apout emulator would be able to recompile the ‘last1120c’ compiler, and so it turned out to be. And, using the newly-compiled executables *c0* and *c1* built from ‘last1120c’, the compiler was able to recompile itself.

The ‘prestruct-c’ compiler presented a much harder problem: some of the source code was missing, particularly the hand-coded tables used to convert the compiler’s internal intermediate form into final assembly code. This seemed at first an insurmountable problem, but after exploring several dead ends a solution was found. The hand-coded tables from the ‘last1120c’ compiler were borrowed and, with a small number of changes, the hybrid source code was able to be compiled by the ‘last1120c’ compiler, and then to compile itself.

6 1st Edition UNIX Kernel: 1971

Alas, with the above DECTapes fully explored, there seemed to be no earlier UNIX artifacts except Ritchie’s fragmentary code listings on paper. Then in 2006, Al Kossow from the Computer History Museum unearthed and scanned in a document by T. R. Bashkow entitled “Study of UNIX”, dated September 1972 [1]; this covers “the structure, functional components and internal operation of the system”. Included along with the study was what appeared to be a complete listing of an assembly version of the UNIX kernel. A second document unearthed contained the handwritten notes made in preparation of Bashkow’s study; dates within this document

indicate that the analysis of the UNIX kernel began in January 1972, implying that the kernel being studied was the 1st Edition UNIX kernel.

The idea of restoring the listing of the 1st Edition kernel to working order seemed impossible: there was no filesystem on which to store the files, no suitable assembler, no bootstrap code, and no certainty that the user mode binaries on the ‘s2’ tape were compatible with the kernel in the listing; for a while the listing was set aside. Then early in 2008 new enthusiasm for the project was found, and a team of people¹¹ began the restoration work.

The team began by scanning and OCR’ing the kernel listing, creating a separate text document for each page. Each document was manually cross-checked for errors, then combined and rearranged to recreate the original assembly files. The next task was to find a suitable assembler for these files. We found after some trial and error that the 7th Edition assembler could be made to accept the 1st Edition kernel files, but we had to make a few changes to the input files and postprocess the output from the assembler. This raised the issue: how much change can be made to an original artifact when restoring it to working order? We chose to keep the original files intact and create (and annotate) a set of “patch” files which are used to modify the originals for the working restoration. Thus, the original artifact is preserved, and the changes required to bring it back to life are also documented.

The kernel listing and the 1st Edition Programmers Manual indicated that the system required a PDP-11/20 with 24 Kbytes of core, RF-11 and RK03 disks, up to 8 teletypes on a DC-11 interface, and a TC-11 DECTape device. The SIMH PDP-11 simulator was configured to provide this environment. With the kernel assembled into an executable binary, we next had to recreate the boot sequence. Luckily, we were able to side-step this issue by commanding the SIMH simulator to load the executable directly into the system’s memory, initialise some registers and start execution at the correct first instruction.

With fingers crossed, the 1st Edition UNIX kernel was started for the first time for several decades, but after only a few dozen instructions it died. We had forgotten that this early system required the KE11A co-processor. Restoration halted while KE11A support was added to SIMH using the PDP-11/20 processor manual [3]. On the next attempt the kernel ran into an infinite loop, and after studying the code we guessed that the loop on the paper listing was missing a decrement instruction. With this fixed the kernel was able to run in “cold UNIX” mode, which had the task of writing a minimal filesystem onto the RF-11 device along with a number of device files, the *init* program and a minimal command shell.

The filesystem’s format was hand-checked using the format description from the Programmers Manual and determined to be valid, so we pressed on to try booting

the kernel in “warm UNIX” mode. After another couple of kernel source errors were fixed, the 1st Edition UNIX kernel was able to run the *init* program, output a login prompt and invoke a shell on successful *root* login. This was a rather limited success: the early UNIX shell had no metacharacter support (no * expansion), and *echo* was not a built-in. So, with only *init* and *sh* on the filesystem, nothing could be done at the shell prompt. We had several executables from the ‘s2’ tape, but the 1st Edition kernel only supported those with the 0405 header; we took the decision to modify the kernel source to add support for the 0407 executables. Then, with the existing RF-11 filesystem and the Programmers Manual, a standalone program was written to create and populate a filesystem image with the ‘s2’ executables. Now the kernel could be booted to a usable system with nearly all of the documented 1st Edition system tools, editors, document processing tools and programming languages.

We now had the system running in single-user mode, but the kernel listing showed that it normally ran in multi-user mode: there was only one process in memory at any time; all other processes were kept on swap. Our attempts to configure the system for multi-user mode simply resulted in the system ‘hanging’ at boot time. Again, a hardware configuration deficiency was found: the SIMH simulator had no support for the DC-11 serial interface device. Using the 1972 PDP-11 peripherals handbook [4] we added DC-11 support to SIMH, and finally brought 1st Edition UNIX up into multi-user mode. The restoration of the kernel was complete.

While the C language did not exist for the 1st Edition of UNIX, there was a C compiler in existence by the time of the 2nd Edition [9]. We had the ‘last1120c’ C compiler source code and working executables, but to run them the restored kernel & filesystem needed to be modified to provide a 16 Kbyte process address space and 16 Kbyte swap areas on the disk. With these modifications the restored system was able to run the C compiler, and the C compiler was able to recompile itself.

7 Lessons Learned

From successfully completing the restoration of the above UNIX software artifacts, we have learned several lessons about the craft of software restoration:

Restoration is only possible with adequate documentation. This not only includes user manuals, but manuals for system calls, libraries, file and storage structures, documentation on how to configure and boot systems, and technically solid hardware manuals.

Comments and documentation are often misleading. Though documentation is required, it is not always accurate or complete. A restorer must treat all documentation as dubious, and look for independent sources

which corroborate each other.

Restoration is only possible with a working environment. All software requires an environment in which to execute. User mode executables require a CPU to run instructions, some memory, and access to a set of system calls: this is what emulators like Wine and Apout provide. Operating systems require a suitable hardware environment, real or simulated. If the correct environment cannot be recreated, restoration cannot proceed.

Restoration from source requires a working compilation environment. Source code is tantalizingly close to being executable, but without a compiler or assembler that can recognise the source and produce the executable, the source code is just a collection of bits.

Any restoration will affect the purity of the original artifact. It is next to impossible to recreate the environment required to run a software artifact older than a decade, as the hardware and supporting software often no longer exist. It is therefore usually necessary to modify both the software artifact and the recreated environment so that they are compatible. When this occurs, it is imperative to preserve the purity of the original artifact, and copy & “patch” it to perform a working restoration.

Simulated hardware is infinitely easier to obtain, configure and diagnose than real hardware. Tools like SIMH can be configured to simulate a vast combination of different CPUs, memory sizes and peripherals. They can be easily single-stepped, and register & memory dumps can be taken at any point. This allows the diagnosis of errant software behaviour much more quickly than with real hardware.

Never underestimate the ‘packrat’ nature of computer enthusiasts. Artifacts that appear to be lost are often safely tucked away in a box in someone’s basement. The art is to find the individual who has that box. The formation of a loose group of interested enthusiasts, TUHS, has helped immensely to unearth many hidden treasures. And professional organisations such as the Computer History Museum are vital if the computing industry wants to preserve a detailed record of its past.

In conclusion, the restoration of some of the earliest software artifacts from the development of UNIX has been time-consuming, frustrating but most importantly extremely rewarding. It is now more important than ever to begin to preserve computing history, not as a collection of “stuffed exhibits”, but to keep old systems running as was intended by their designers.

8 Acknowledgments

None of the work described in this paper would have been possible without the generosity of the members of the UNIX Heritage Society, who donated software, docu-

mentation, anecdotes & memories, provided suggestions & insights, and gave time to lobby the powers that be to place the early UNIX systems under an open source license. Dennis Ritchie in particular has not only provided artifacts, memories and advice, but has also encouraged and mentored the restoration process: to him I owe a profound thanks. Finally, we are all indebted to Ken Thompson, Dennis Ritchie, the researchers at Bell Labs and the cast of thousands who made UNIX into such a powerful, sophisticated and pleasant system to use.

References

- [1] BASHKOW, T. A Study of the UNIX Operating System, Sep 1972. http://www.bitsavers.org/pdf/bellLabs/unix/PreliminaryUnixImplementationDocument_Jun72.pdf.
- [2] BURNETT, M., AND SUPNIK, R. Preserving Computing’s Past: Restoration and Simulation. *Digital Technical Journal* (1996), 23–38.
- [3] DEC. PDP-11/20 Processor Handbook, 1971. http://www.bitsavers.org/pdf/dec/pdp11/handbooks/PDP1120_Handbook_1972.pdf.
- [4] DEC. PDP-11 Peripherals Handbook, 1972. http://www.bitsavers.org/pdf/dec/pdp11/handbooks/PDP11_PeripheralsHbk_1972.pdf.
- [5] LIBES, D., AND RESSLER, S. *Life with UNIX*. Prentice Hall, 1989.
- [6] LIONS, J. *A Commentary on UNIX 6th Edition with Source Code*. Peer-to-Peer Communications, 1996.
- [7] MAHONEY, M. The UNIX Oral History Project, 1989. <http://www.princeton.edu/~mike/exptape.htm>.
- [8] RITCHIE, D. M. The Evolution of the UNIX Time-Sharing System. *BSTJ* 63, 8 (1984), 1577–1594.
- [9] RITCHIE, D. M. The Development of the C Language. In *Proceedings of the Second History of Programming Languages Conference* (Apr 1993).
- [10] SALUS, P. H. *A Quarter Century of UNIX*. Addison Wesley, 1994.
- [11] TOOMEY, W. Saving UNIX from /dev/null. In *Proceedings of the AUUG Open Source Conference* (1999).

Notes

¹UNIX is a registered trademark of The Open Group.

²See the excellent UNIX family tree by Éric Lévénez at <http://www.levenez.com/unix/>

³See <http://www.tuhs.org>

⁴Old SCO, as against the SCO Group (TSG).

⁵Early UNIX source code has very spartan commenting.

⁶See page 4 of http://bitsavers.org/pdf/bellLabs/unix/Unix_Users_Talk_Notes_Jan73.pdf

⁷1st Edition UNIX used an 0405 a.out signature. 2nd Edition UNIX changed to an 0407 signature, indicating a slightly different format.

⁸See <http://www.tuhs.org/Archive/PDP-11/Emulators/Apout/>

⁹See <http://minnie.tuhs.org/Programs/Ctcompare/>

¹⁰See <http://cm.bell-labs.com/cm/cs/who/dmr/primevalC.html>

¹¹The team was led by Tim Newsham & Warren Toomey, along with Johan Beiser, Tim Bradshaw, Brantley Coile, Christian David, Alex Garbutt, Hellwig Geisse, Cyrille Lefevre, Ralph Logan, James Markevitch, Doug Merritt and Brad Parker.

Block Management in Solid-State Devices

Abhishek Rajimwale⁺, Vijayan Prabhakaran^{*}, John D. Davis^{*}

⁺*University of Wisconsin, Madison*

^{*}*Microsoft Research, Silicon Valley*

Abstract

*Solid-state devices (SSDs) have the potential to replace traditional hard disk drives (HDDs) as the *de facto* storage medium. Unfortunately, there are several decades of spinning-media assumptions embedded in the software stack as an “unwritten contract” [20]. In this paper, we revisit these system-level assumptions in light of SSDs and find that several of them are invalidated by SSDs, breaking the unwritten contract and resulting in poor performance and lifetime. The underlying cause is the incorrect division of labor between file systems and storage. Block management must be removed from the file system and delegated to the SSD to prevent further accumulation of storage-specific assumptions. We find that object-based storage is an appropriate way to achieve this.*

1 Introduction

Storage systems export a simple abstraction of a linear block-level interface that has worked well for cases ranging from a single disk to the aggregation of disks such as RAID arrays and logical volumes. In fact, the simplicity of the interface has helped to hide the complexity of the underlying device from higher-level systems.

Unfortunately, the interface has also hidden device-specific details, so that the file system is forced to make assumptions about the underlying storage, referred to by Schlosser and Ganger as an “unwritten contract” [20]. Not surprisingly, most of these assumptions are regarding block management such as allocation, layout, scheduling, and cleaning, all of which could benefit from device-specific knowledge. For example, file systems assume that random accesses are much slower than sequential ones, and hence the block management layer is optimized for this. While this assumption is valid for disks, when the properties of the underlying device change, such assumptions may either hold or fail. For

example, for MEMS-based storage, Schlosser *et al.* find that the existing abstractions are mostly valid [20].

We reexamine the existing storage abstraction and the resulting assumptions in light of solid-state devices (SSDs). SSDs are different from disk drives in many aspects such as unique semiconductor properties, internal architecture, and controller firmware, which affect the performance, reliability, lifetime, power, and security properties of the SSDs. Overall, SSDs are substantially complex and self-managing and require more information than is provided by the standard storage interface.

To find out if the disk-specific assumptions hold for SSDs, we list six system-level assumptions (three of which are from the unwritten contract as originally stated) and explain how each of them fail for SSDs, resulting in poor performance and lifetime. The underlying problem is the incorrect division of labor: file systems perform block management, which for a device such as an SSD is best done internally because of its knowledge of intricate device-specific properties, policies, and algorithms.

A more expressive interface such as object-based storage (OSD) [10, 11, 22] can improve the current state of the art. First, OSD delegates finer details of block management to the SSD, thereby preventing any new storage-specific assumptions. Second, OSD expresses the intentions of the higher layers clearly, thereby improving the internal SSD operations.

2 Background

Storage Interface. Storage access protocols such as SCSI and ATA/IDE export a narrow, block-based interface with simple `read` and `write` APIs to access the logical block number (LBN) (a 512-byte sector). A storage controller internally maps the LBN to a physical sector, which is hidden and fixed for most cases.

The main disadvantage of the block-based interface is

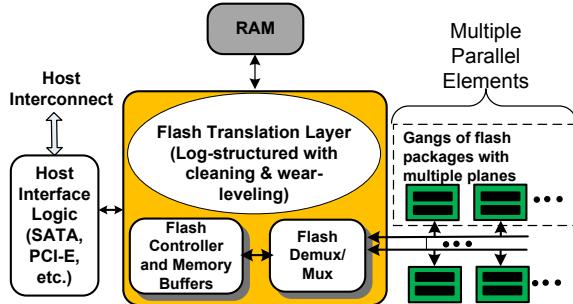


Figure 1: SSD Architecture.

the absence of higher-level semantics. Several previous works note this deficiency and propose more expressive interfaces [5–7, 9]; some even allow programming the storage controller [1, 16, 21]. One approach is to use an object-based interface [10, 11, 22], which exports the abstraction of an object as a collection of bytes. Structures such as trees, tables, files, and directories can be represented as objects, reflecting the higher-level semantics better than a block-based interface; the device controller performs block allocation and layout for the objects.

Solid-State Devices. An SSD consists of a set of flash memory packages that are connected to a controller (Figure 1). Each package has one or more dies; each die has multiple planes, which in turn have many blocks; each block consists of many 4 KB pages [18].

SSDs differ from HDDs on 3 main properties. The first obvious difference is the absence of mechanical moving parts. Second, flash pages are non-overwrite in nature and must be erased before being overwritten. To hide the high erase overhead and create the abstraction of an in-place write, modern SSDs implement a log-structured design [17] in the flash translation layer (FTL) [15]. To uniformly spread the block usage, wear-leveling is also implemented. Finally, SSDs have several layers of parallelism that is dictated by the flash packages and the way they are connected to the controller.

There are two types of NAND flash memory: single-level cell (SLC) and multi-level cell (MLC). SLC flash stores a single bit of data per cell, while MLC flash stores multiple bits per cell. MLC flash has some drawbacks such as shorter lifetime (10K erase cycles vs 100K erase cycles of SLC), slower write, and erase operations.

3 Failed Assumptions

In this section, we discuss the system-level assumptions that fail when applied to SSDs, and the reasons behind these failures. In Table 1, we list the original (1-3) and extended terms of the unwritten contract and state whether they are satisfied or violated by different devices. This list is by no means complete because we

focus only on block-management issues; more assumptions may be added as our experience with SSDs grows. For comparison, we list RAID arrays and MEMS-based storage, but for the rest of the paper we focus only on how the assumptions fail on SSDs.

3.1 Sequential vs. Random

In a disk, the latency and bandwidth of sequential access are several tens of times better than random access. However, on SSDs that use a log-structured FTL [15], both sequential and random writes are likely to take similar time. Table 2 lists the ratio of sequential-to-random bandwidth for an HDD (a Seagate Barracuda 7200.11 drive) and several SSDs. One of the SSDs is simulated ($S4_{slc_sim}$) using the simulator from our previous work [2], while others are real ($S1_{slc}$, $S2_{slc}$, $S3_{slc}$, $S5_{mlc}$). We anonymize the real SSDs because they are engineering samples and pre-production models. To help the reader understand the results better, we specify whether the devices use SLC or MLC flash memory.

From the table, we can observe that SSDs (using SLC or MLC memories) have random-read performance that is only a few times smaller than their sequential-read performance. This is even true for writes on certain SSDs ($S1_{slc}$, $S4_{slc_sim}$, $S5_{mlc}$), but not on all of them; in fact, some of the SSDs ($S2_{slc}$, $S3_{slc}$) have random-write performance that is worse than HDDs. One of the reasons for this poor performance is write amplification, which we will discuss later (§3.4).

From the above results, we can see that the gap between sequential and random accesses is narrowing on SSDs. File systems that run primarily on SSDs must reconsider the need for complex policies to achieve block-level sequentiality. Instead, a file system must focus on higher-level operations such as object management, consistency, and recovery, and move the low-level block management to the SSD, using say, the OSD interface.

3.2 Logical-to-Physical Mapping

The second term of the unwritten contract considers the relation between logical and physical sectors, and understanding it is important for I/O scheduling. On an HDD, nearby LBNs translate well to physical proximity. However, this contract fails on an SSD because of the log-structured design, cleaning, and wear-leveling, all of which make it harder to estimate the location of a logical sector. In fact, the physical location is irrelevant if the ratio of sequential to random accesses approaches 1. This further motivates the conclusion that the file system accesses must be in terms of objects (or parts of objects) and the SSD must handle the low-level sector-specific scheduling.

Contract	Disk	RAID	MEMS	SSD
1. Sequential accesses are much better than random accesses	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i> (flash memory, no mechanical parts)
2. Distant LBNs lead to longer seek times	<i>T</i> †	<i>F</i>	<i>T</i>	<i>F</i> (log-structured writes)
3. LBN spaces can be interchanged	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i> (integration of SLC and MLC memory)
4. Data written is equal to data issued (no write amplification)	<i>T</i>	<i>F</i>	<i>T</i>	<i>F</i> (ganging, striping, larger logical pages)
5. Media does not wear down	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i> (semiconductor properties)
6. Storage devices are passive with little background activity	<i>T</i> †	<i>F</i>	<i>T</i>	<i>F</i> (cleaning and wear-leveling)

Table 1: **Unwritten Contract.** Terms of the unwritten contract and whether they are satisfied (*T*) or not (*F*) by various devices; a † means that the contract is only approximately satisfied because the device has grown more complex. For SSDs, a brief reason is also given.

Device	Read			Write		
	Seq	Rand	Ratio	Seq	Rand	Ratio
HDD	86.2	0.6	143.7	86.8	1.3	66.8
<i>S1_{slc}</i>	205.6	18.7	11.0	169.4	53.8	3.1
<i>S2_{slc}</i>	40.3	4.4	9.2	32.8	0.1	328.0
<i>S3_{slc}</i>	72.5	29.9	2.4	75.8	0.5	151.6
<i>S4_{slc,sim}</i>	30.5	29.1	1.1	24.4	18.4	1.3
<i>S5_{mlc}</i>	68.3	21.3	3.2	22.5	15.3	1.5

Table 2: **Ratio of Sequential to Random Bandwidth.**

We performed a preliminary analysis with a new algorithm for SSD, called shortest wait time first (SWTF), which uses the queue wait times of all the parallel elements in an SSD and schedules an I/O that has the shortest wait time. On a synthetic workload that issues random I/Os (with 2/3 reads and 1/3 writes), we found that SWTF improves the response time by about 8% when compared to FCFS. More thorough analysis is required to find the effectiveness of such SSD-specific algorithms.

3.3 Interchangeable Address Space

The third term of the contract assumes that the logical address space is uniformly spread over the device. This is invalidated by disks because of zoned recording, where the outermost tracks accommodate more logical pages than innermost ones; that is, outer-track bandwidth is greater than the inner-track bandwidth. Today, SSDs are homogeneous, using only a single type of memory (either SLC or MLC), keeping the contract valid. However, we believe that in the future, SSDs might be constructed with multiple types of memories (SLC/MLC). In such systems, this contract will be violated because MLC-type memories can hold more data and have different timing characteristics than SLC. Such heterogeneity in the address space can be better utilized if the device performs block allocation for higher-level objects. For example, an SSD can choose to co-locate all the data belonging to a root object in SLC memory for faster access.

3.4 Write Amplification

Operating systems typically assume that the time taken to complete an I/O is proportional to the I/O size. However, in an SSD, writes may be amplified into a larger

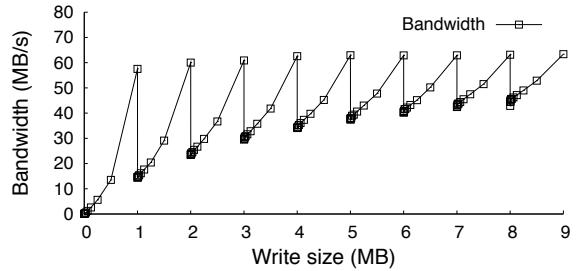


Figure 2: **Write Amplification.** In *S2_{slc}*, maximum bandwidth is achieved when the write size aligns with the stripe size (1 MB).

I/O due to several reasons: first, when the logical page is larger than the physical page size; second, when a write is issued in-place using a read-modify-erase-write cycle. Write amplification is not a new phenomenon; it happens on RAID arrays that need to update parity blocks.

We measured the effect of write amplification on one of the engineering samples (a low-end SSD, *S2_{slc}*); Figure 2 shows the results. We plot the bandwidth against the write size. One can observe that the bandwidth is poor on small write sizes (e.g., 512 bytes). As we increased the write size, the bandwidth improved and reached its maximum at 1 MB (the SSD’s stripe size). As we increased the write size further (e.g., 1 MB + 512 bytes), the bandwidth again dropped, and this behavior repeated to give a saw-tooth pattern. We believe that this behavior is due to striping the logical page across a gang of flash packages that share the buses [2]. A similar saw-tooth pattern was noticed by Schindler *et al.* for disk drives on track-aligned accesses [19]. However, in their case it was due to the effect of track switches and rotational latencies. It is important to note that write caches might not always mask the write amplification; for example, *S3_{slc}* has a write buffer cache of 16 MB, but it is ineffective in masking the write amplifications, as can be seen from the random-write performance in Table 2.

Write amplification can be reduced by merging writes and aligning them to stripe sizes. Since it is harder to estimate the stripe size and alignment boundaries from a file system (especially in the presence of a write cache

Probability of sequential access	0	0.2	0.4	0.6	0.8
Unaligned	10.6	10.6	10.5	10.2	10.5
Aligned	10.6	10.4	8.9	7.6	5.6

Table 3: **Improved Response Time with Write Alignment.** Average I/O response time (in ms) for unaligned and stripe-aligned 4 KB writes with varying degrees of sequentiality.

	Postmark	TPCC	Exchange	IOzone
Improvement (%)	1.15	3.08	4.89	36.54

Table 4: **Macro Benchmarks with Stripe-aligned Writes.**

and background activity), an SSD must be responsible for sector allocation and layout according to the stripe sizes. Table 3 shows results from stripe-aligned and unaligned writes. We simulated a 32 GB SSD with one gang of eight 4 GB flash packages. A single 32 KB logical page spanned over all the packages. We ran a synthetic workload that issued a stream of writes with varying degrees of sequentiality. We compared two schemes: one, issuing the writes as they arrive; two, merging and aligning writes on logical page boundaries. On a completely random workload, both schemes worked similarly because of the small chance to merge the writes into stripe sizes. As the sequentiality increased, aligning writes paid off well, resulting in an improvement of over 50%. Table 4 presents the improvement in response time for various workload traces. Of all the workloads, IOzone benefits the most (over 36% improvement) due to its large write sizes.

3.5 Block Wear

File systems assume that the media wear-down does not depend on the number of writes to particular sectors. However, flash memory blocks have limited erase cycles before wearing out. Therefore, mid-range and high-end SSDs implement cleaning and wear-leveling to uniformly spread the wear-down of blocks.

SSDs clean by retaining the most recent version of *all* the logical pages, including those that have been released by the file system, leading to a lot of useless activity. The effectiveness of cleaning and wear-leveling can be improved by using file-system-level semantic knowledge, specifically the block allocation status, which is not available to an SSD.

An SSD can use the block allocation status to implement *informed cleaning and wear leveling* that avoids retaining the free pages. We used our SSD simulator to analyze the benefits of informed cleaning by running block-level traces that contain read, write, and block-free operations. The traces were collected by running the

Transactions	5000	6000	7000	8000
Relative pages moved	0.31	0.25	0.35	0.50
Relative cleaning time	0.69	0.60	0.63	0.69

Table 5: **Improved Cleaning with Free-Page Information.** The table shows decrease in pages moved and cleaning time with free-page information relative to the default SSD (i.e., without free-page information). The actual numbers of pages moved (in 1000s) for the default SSD are, 88159, 155465, 217130, and 284409, for 5K to 8K transactions. The actual cleaning times (in seconds) for the default SSD are, 49147, 71975, 93569, and 116185 for 5K to 8K transactions.

Postmark benchmark [14] on a pseudo-device driver that uses Linux Ext3 knowledge to identify the free sectors. The SSD simulator was modified such that the cleaning and wear-leveling logic disregard the flash pages corresponding to the free logical pages. To the best of our knowledge, this is the first study to measure the effect of free-page knowledge in SSD cleaning.

Table 5 shows the improvements in cleaning in terms of the number of pages that need to be reclaimed and the cleaning time for an 8 GB SSD; both measures are shown relative to the default SSD that does not use the free-page information. We observe that informed cleaning reduces the number of reclaimable pages by at most about one-half. Informed cleaning reduces the cleaning time by 30-40%, which can improve the overall running time by about 3-4%. However, the improvements are workload-dependent.

3.6 Background Activity

Storage systems are assumed to be passive and to act only when a request is issued from a higher-level software layer. While this is true on a single HDD, SSDs perform a considerable amount of background activity due to cleaning and wear-leveling. Therefore, it becomes hard to predict the I/O latency; for example, it is hard to guarantee QoS on a system with SSDs because the host has no control over when the SSD engages in background activities. This is especially true if the SSD is full and the degree of internal fragmentation is high [4]. The background activity can be controlled by informing the SSD about I/O priorities or by marking certain objects as high-priority. For example, an SSD can provide preferential treatment to high-priority objects or I/Os by delaying its background activity.

We modified the cleaning logic of our SSD simulator to be aware of request priorities. If there are no outstanding priority requests, cleaning starts when the number of free pages falls below a *low* threshold. However, if there are priority requests, cleaning is postponed until the number of free pages falls below a *critical* threshold. We call this *priority-aware* and compare it with a *priority-*

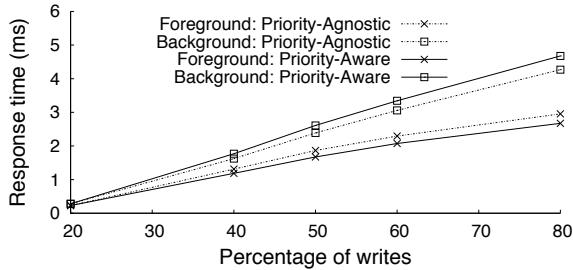


Figure 3: **Priority-Aware Cleaning.** Priority-aware cleaning improves the foreground I/O response time by postponing cleaning; in contrast, priority-agnostic cleaning deteriorates the foreground I/Os.

Writes (%)	20	40	50	60	80
Improvement (%)	0	9.56	10.27	9.61	9.47

Table 6: **Response Time Improvement From Priority-Aware Cleaning.** When the percentage of writes is small (less than 40%), cleaning does not happen frequently and hence foreground requests are not affected (and therefore no improvement).

agnostic scheme, which starts cleaning at the low threshold irrespective of the outstanding requests. Note that when there is no priority information available, priority-agnostic is the default technique to use.

We evaluated a 32 GB SSD using synthetic benchmarks with request inter-arrival times uniformly distributed between 0 and 0.1 ms. The fraction of priority requests was set to 10%; critical and low thresholds were fixed at 2% and 5% of free pages. In Figure 3, we plot the response time of priority requests (marked as foreground) and non-priority requests (marked as background). Table 6 shows the corresponding improvement in response time for priority requests. We observe that under the priority-aware scheme, the response time of foreground requests improve by about 10%. However, the cost of this improvement is reflected on the background I/Os, whose response time increased as well.

3.7 Object-based Storage and SSD

As storage devices grow more complex, assumptions made by higher layers fail. We believe that certain functionalities, specifically those related to block management, are more appropriately handled by the device controller with its intricate knowledge of the inner workings of the device. However, to perform the block management correctly and efficiently, devices must also understand the high-level intentions (semantics) behind the simple reads and writes.

Broadly, there are two ways by which the device can obtain more information: explicitly, through new or

modified interfaces; or implicitly, by using reverse engineering techniques. Since reverse engineering techniques can add more complexity to the device firmware and do not minimize the functionalities at higher layers, we focus only on explicit approaches. Existing interfaces can be patched with additional commands to convey the operation semantics. For example, the TRIM command has been proposed to add file delete notifications to the ATA interface [8]. While this approach offers the least resistance in the device-interface evolution, it still operates on the block level, thereby letting the file system perform the block management. Moreover, existing interfaces may not provide sufficient extensions for new commands. In such cases, new interfaces such as NVMHCI have been proposed [13]. However, while NVMHCI conveys more information than traditional SCSI/ATA, it still lets the higher layers manage and operate at the block level. We believe that OSD interface provides a nice alternative by conveying more information and letting the device handle low-level operations.

For several of the aforementioned contract violations, an OSD provides a better alternative. For example, a file system should operate on objects and let the device handle the logical to physical mapping, sequential-random accesses to (parts of) objects, and stripe-aligned accesses. Additionally, an SSD can use the OSD interface and manage the space for objects (including the allocation and release of pages to objects) in order to implement informed cleaning. An additional benefit of using an OSD is that object attributes can be set to convey read-only data, which could be used for cold data placement during wear-leveling. Finally, I/Os to objects can be marked with a priority to schedule them appropriately with background activities.

4 Related Work

Several previous researchers have noted the need for more expressive storage interfaces for disks [1, 5, 6, 9, 16, 21], RAID arrays [7], and MEMS-based devices [12, 20]. Among these, the most closely related work is by Schlosser and Ganger, which examines the OS assumptions in the context of MEMS-based devices [20]. They list the first three terms of the unwritten contract and show how MEMS-based devices obey them, obviating the need for new interfaces or algorithms. In another related paper, Ajwani *et al.* characterize a variety of SSDs and argue for new algorithms for SSDs and hybrid devices [3]. However, they still use the block-level interface. We argue for a new, richer interface.

New interface specifications are being proposed for SSDs, like NVMHCI [13] and TRIM [8], but they still let the higher layers manage the blocks, resulting in most of the problems we discussed earlier. Using the OSD inter-

face provides a clean separation between the file system and block management operations, enabling the SSD to handle them optimally.

5 Conclusion

Over the past 5 decades, OS and storage systems have evolved independently across a narrow and fixed storage interface. One of the side effects of this evolution is the accumulation of device-specific assumptions in the storage stack, specifically in the block management layer. Unless the block management is removed from the file system and delegated to the storage, such assumptions are likely to carry over and grow in the next generation of storage devices as well. SSDs are evolving and have the potential to become the ubiquitous storage media. As our initial results have shown, it is time we switch from the narrow, block-based interface to a richer object-based storage to improve the performance and longevity.

6 Acknowledgments

We thank our shepherd, Geoff Kuennen, the anonymous reviewers, Nathan Obr from Microsoft Device and Storage Technologies, and the members of Microsoft Research Silicon Valley for their detailed feedback and excellent suggestions on our paper.

References

- [1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.
- [3] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the Performance of Flash Memory Storage Devices and Its Impact on Algorithm Design. In *Experimental Algorithms*, pages 208–219. Springer Berlin / Heidelberg, 2008.
- [4] L. Bouganim, B. Jonsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR'09)*, 2009.
- [5] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.
- [6] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 15–28, December 1993.
- [7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 177–190, June 2002.
- [8] Frank Shu. Notification of Deleted Data Proposal for ATA8-ACS2. http://t13.org/Documents/UploadedDocuments/docs2007/e07154r0-Notification_for_Deleted_Data_Proposal_for_ATA-ACS2.doc, 2007.
- [9] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.
- [10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 92–103, October 1998.
- [11] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie Mellon University, 1997.
- [12] J. Griffin, S. Schlosser, G. Ganger, and D. Nagle. Operating Systems Management of MEMS-based Storage Devices. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.
- [13] Intel Corporation. Non-Volatile Memory Host Controller Interface Specification. <http://www.intel.com/standards/nvnhci/index.htm>, 2008.
- [14] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.
- [15] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, 2008.
- [16] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proceedings of the 24th International Conference on Very Large Databases*, August 1998.
- [17] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [18] Samsung Corporation. K9XXG08XXM Flash Memory Specification. http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/NANDFlash/SLC_LargeBlock/8Gbit/K9F8G08U0M_ds_k9f8g08x0m_rev10.pdf, 2007.
- [19] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 259–274, 2002.
- [20] S. W. Schlosser and G. R. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole? In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, pages 87–100, April 2004.
- [21] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Evolving RPC for active storage. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 264–276, October 2002.
- [22] R. O. Weber. SCSI Object-Based Storage Device Commands (OSD). Technical report, T10 Technical Committee, July 2004.

Linux Kernel Developer Responses to Static Analysis Bug Reports

Philip J. Guo and Dawson Engler
Stanford University

Abstract

We present a study of how Linux kernel developers respond to bug reports issued by a static analysis tool. We found that developers prefer to triage reports in younger, smaller, and more actively-maintained files (§2), first address easy-to-fix bugs and defer difficult (but possibly critical) bugs (§3), and triage bugs in batches rather than individually (§4). Also, although automated tools cannot find many types of bugs, they can be effective at directing developers' attentions towards parts of the codebase that contain up to 3X more user-reported bugs (§5).

Our insights into developer attitudes towards static analysis tools allow us to make suggestions for improving their usability and effectiveness. We feel that it could be effective to run static analysis tools continuously while programming and before committing code, to rank reports so that those most likely to be triaged are shown to developers first, to show the easiest reports to new developers, to perform deeper analysis on more actively-maintained code, and to use reports as indirect indicators of code quality and importance.

1 Methodology

We used two datasets for quantitative analysis: static analysis bug reports and source code revision history.

We obtained static analysis results from the Coverity Scan project [5], which uses a commercial tool called Coverity Prevent to find bugs in open source C, C++, and Java projects. Coverity reports its results in an online bug database and tracks if and when developers triage, verify, and fix those bugs. When a developer triages a bug report, he/she tries to determine the veracity of the report and then changes its status in the database from *un-triaged* to *true bug*, *false positive*, or, if he/she gives up without reaching a definitive conclusion, to *unsure*.

We obtained 2,125 bug reports produced by scans run between February 2006 and December 2007. Each report pinpoints a potential bug within a .c source file in

the Linux kernel codebase. The *initial scan* on Feb 24, 2006 created 981 reports, and the *76 subsequent scans* run periodically between then and December 2007 created 1,144 additional reports.

To get development histories for files affected by Coverity Scan reports, we mined version control data from the BitKeeper and GIT Linux kernel source code management repositories, spanning February 2002 to December 2007. We recorded when each file was added to the codebase and detailed information about each committed patch (patch size, date, author, files affected).

To corroborate our quantitative findings and to add qualitative insights, we sent out an informal email questionnaire to the primary Linux kernel developers mailing list. In that questionnaire [7], we stated each of our findings (worded identically to how it appears in this paper) and asked developers to present reasons why they agreed or disagreed with it based upon their experiences and intuitions. We received 4 responses and will quote their authors as developers A, B, C, and D due to requests for anonymity. We got the opinions of some veteran developers: Developer A has triaged the most Coverity Scan reports out of all 26 developers who have triaged reports, and developers A and B are both in the 99th percentile in terms of numbers of patches written for the Linux kernel.

2 Which reports are likely to be triaged?

Result 1: Checker type is the most important factor in determining whether a bug report will be triaged

Coverity Prevent checks for a dozen types of generic C code bugs, such as buffer overflows and null pointer dereferences (the Coverity Open Source Report [5] describes all types in detail).

Table 1 shows percents of triaged reports (*triage rate*), which vary greatly across checker types. All developers who responded to our questionnaire agreed that checker

Checker type	# reports	Total		% of triaged reports in	
		% triaged	relative FP	initial scan	all subsequent scans
dynamic buffer overrun	6	100%	3	★	★
read of uninitialized values	64	86%	5	84%	88%
dead code	266	82%	6	71%	88%
static buffer overrun	288	79%	8	74%	82%
unsafe use before negative test	13	69%	9	★	★
type/allocation size mismatch	5	60%	1	★	★
unsafe use before null test	256	57%	2	65%	48%
resource leak	302	54%	4	52%	56%
null pointer dereference	505	51%	7	54%	46%
unsafe use of null return value	153	50%	12	72% [†]	37% [†]
use resource after free	225	49%	11	72% [†]	41% [†]
unsafe use of negative return value	42	38%	10	36%	43%
Total	2,125	61%		63%	59%

Table 1: Coverity Scan reports by checker type, sorted by triage rate (“% triaged”). The “relative FP” for each checker is its false positive (FP) rate relative to all other checkers (1 means *lowest* false positive rate, 12 means *highest*). The ★ symbol is for checkers with too few reports to make meaningful differences between initial and subsequent scans.

type most strongly determines whether they triage a report; the one who triaged the most reports emphasized,

“I always sort the reports by report type and don’t care which files they are in.” (Dev A)

We corroborated these intuitions by building a predictive model using all factors in this section and noting that checker type was by far the strongest predictor of whether a report would be triaged; we describe our model’s details in a separate technical report [8].

One reason why reports from certain checker types are triaged more frequently is that they find more severe bugs. Considering the top 4 checkers in Table 1, buffer overruns lead to security vulnerabilities, reads of uninitialized values lead to non-deterministic failures, and dead code bugs often indicate serious logic errors arising from the developer’s misunderstanding of what ought to be able to execute under which exact (sometimes multiply-nested) conditions.

In contrast, reports from certain checkers are triaged less frequently because they are harder to diagnose:

“I have looked at a few coverity defects and skipped over them because a) they looked too hard to diagnose b) They looked like false positives but I didn’t have enough knowledge about the code to be positive” (Dev C)

Specifically, the more code a developer must read when investigating a bug report, the more likely he/she will skip that report. Checkers for unsafe uses of null/negative function return values had low triage rates,

perhaps because they require developers to look inter-procedurally to assess whether the called function can return a null or negative value during actual execution.

Also, developers are reluctant to triage reports from checkers whose reports they have marked as false positives. There is an inverse correlation between false positive rate and triage rate: a Spearman’s rank correlation¹ of -0.49. In particular, the 3 checkers with the highest false positive rates also had the lowest triage rates.

To show that triage rates don’t vary much across scans, we calculated separate rates for the initial scan and for all 76 subsequent scans taken together. The relative rankings of checkers remained fairly consistent across the two populations, with a Spearman’s rank correlation of 0.79. For this calculation, we excluded the 2 checkers with the highest false positive rates ([†] symbol) due to their aberrant drop-offs, which Result 8 will discuss, and checkers with too few reports (★ symbol).

Result 2: Bug reports in younger files are more likely to be triaged

From 2002 to 2007, the Linux codebase grew linearly by 173 new files each month (on average, 326 files were added and 153 deleted each month). The linear regression line (not pictured here) fits almost perfectly, with adjusted R-squared of 0.992 (1.0 is a perfect positive linear correlation). Files are typically quite active during their first year of life, receiving up to twice as many patches during that year than during their subsequent

¹Spearman’s rank correlation test determines the direction and consistency of correlation between two variables, returning a value between -1 and 1. 1 means perfect positive (non-linear) correlation, 0 means no correlation, and -1 means perfect negative correlation.

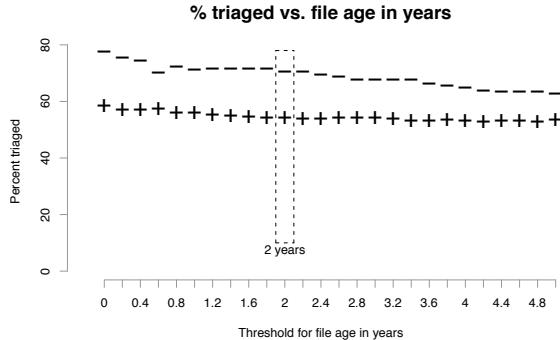


Figure 1: Percent of reports triaged in all files older (+) and younger (−) than selected thresholds. At 2 years, there is a 31% relative difference in triage rates between old and young files (54% vs. 71%).

years. Developers are more interested in bug reports for these younger, more active files than for older files:

“My gut feeling says that [result] is probably right. More often the people involved in creating those [younger] files will still be active kernel developers, and still interested in the area those files cover.” (Dev B)

If we split files into two groups by their age at the time of each bug report using some reasonable cutoff between “young” and “old” files (say, 2 years) and then count the numbers of triaged and un-triaged reports affecting files within each group, we find that 71% of bug reports affecting young files are triaged, versus only 54% of reports affecting old files. We used a chi-square test² to establish statistical significance: The probability a difference of this magnitude appearing by chance is nearly zero ($p = 3.8 \times 10^{-13}$).

However, the choice of 2 years as a threshold is somewhat arbitrary and could have been made to maximize the apparent disparity in triage rates, so we performed the same calculations for a wide range of age thresholds and plotted the triage rates for old and young files with each threshold along the x-axis in Figure 1. For all choices of thresholds within the range of our dataset (points along the x-axis), older files (marked by +) had a lower triage rate than younger files (marked by −). The differences are all significant with $p < 0.01$ in a chi-square test.

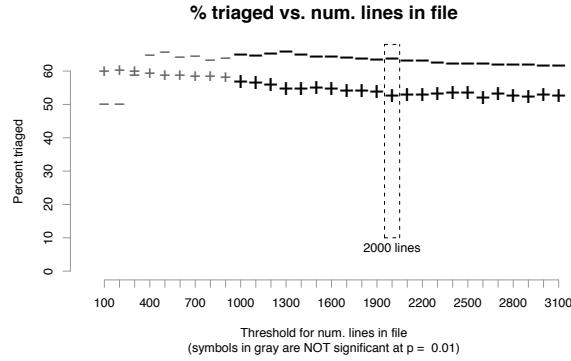


Figure 2: Percent of reports triaged in files larger (+) and smaller (−) than a threshold number of lines. At 2000 lines, there is a 21% relative difference (53% vs. 64%).

Result 3: Bug reports in smaller files are more likely to be triaged

Our dataset consists of 14,646 .c source files added between 2002 and 2007; out of these, 68% had less than 500 lines, 86% less than 1000 lines, and 95% less than 2000 lines. Figure 2 shows that, for most thresholds, reports in smaller files (marked with −) are more likely to be triaged than those in larger files (marked with +). The disparity is not as large as that for age, though; thresholds below 1000 lines result in differences that fail to achieve statistical significance at $p = 0.01$ in a chi-square test (denoted by grayed-out symbols in Figure 2).

One developer offered these possible causes:

“Possibly, perhaps due to the buried in warnings syndrome. Perhaps also because smaller files are easier to modify.” (Dev C)

Developers can be overwhelmed when viewing too many reports at once, becoming “buried in warnings”. Smaller files usually contain fewer reports, so developers might be more willing to triage reports in those files.

Also, smaller files are easier to understand and modify since they usually have fewer functions and implement simpler and more self-contained features. When Phang et al. performed a user study where subjects triaged bug reports from a static analysis tool, they found that complexity of program paths and inter-procedural control flow made the triaging task more difficult [12].

Result 4: Triage rates vary across kernel sub-systems

Table 2 shows triage rates for files in different sub-systems (split by top-level directory). As expected, core kernel files had the highest triage rate (in addition to files in the kernel/ directory, we also included arch/ since it contains many architecture-specific core kernel

²The chi-square test for equality of proportions can determine whether the proportion of occurrences of one binary variable (e.g., will a particular report be triaged?) depends on the value of another binary variable (e.g., is file age less than 2 years?). This test produces a p-value that expresses the probability a purported difference in proportions could have arisen by chance; typically, $p < 0.01$ indicates statistical significance.

Sub-system:	# reports	% triaged	med. days
core kernel	79	67%	1
drivers	1,329	64%	14
memory	16	63%	3
filesystems	309	59%	13.5
networking	341	47%	14

Table 2: Percent of reports triaged by sub-system, and median number of days it took to triage each report.

files). Also, reports in core kernel and memory management (`mm/`) code were triaged much faster than reports in larger sub-systems that contain some more obscure code (e.g., there are numerous rarely-deployed drivers, filesystems, and network protocols). Surprisingly, driver bugs had the second highest triage rate; many drivers are rarely used and aren’t actively maintained, so we expected far fewer driver bugs to be triaged.

Independence of factors: A problem that arises when presenting a series of single-variable correlations (like we’ve done in this section for factors that correlate with triage rates) is that these factors might be cross-correlated, thereby diminishing the validity of the results.

To show that our factors have *independent effects*, we built a logistic regression model³ to predict whether particular Coverity Scan reports will be triaged; we describe our model’s details in a technical report [8]. We used the four factors in this section — checker type, file age, file size, and sub-system — in our model. We determined that all factors had independent effects by incrementally adding each one to an empty model and observing that the model’s deviance (error) decreases by a statistically significant amount for all added factors (a standard technique called *Analysis of Deviance*). Checker type was the strongest factor because it decreased our model’s deviance by the greatest amount.

Redundant factors: Other factors also significantly correlated with triage rates, most notably the number of patches and number of developers modifying the affected file. However, both are highly dependent on file age: Intuitively, the longer a file has been alive, the more opportunities it has for receiving patches and for having more developers. Since the kernel developers in our questionnaire responded most favorably to file age as a determiner for whether reports are triaged and did not prefer the other two related factors as much, we used file age in our model and discarded the other two factors.

³A logistic regression model aims to predict the value of a binary variable (e.g., *will a particular report be triaged?*) using a combination of numerical (e.g., *file age*) and categorical factors (e.g., *checker type*).

	Unsure	1 month	3 months	6 months	1 year
\geq		18%	23%	36%	54%
<		18%	17%	17%	17%
<i>p-value</i>		0.96	0.07	~ 0	~ 0
	True Bug	1 month	3 months	6 months	1 year
\geq		27%	23%	18%	11%
<		35%	35%	34%	33%
<i>p-value</i>		0.05	0.008	0.02	0.06

Table 3: Percent of *triaged* reports that were marked as *unsure* (top) and as *true bug* (bottom), split by time it took to triage each report (with chi-square *p-values*).

Discussion: Static analysis tools can produce thousands of bug reports, but those reports are useless unless developers triage them. Tool makers can use factors like those described in this section to build models to predict the likelihood that particular future reports will be triaged. The tools can then first show developers reports that are most likely to be triaged. This type of ranking system is currently deployed at Google [3, 13].

3 Which reports are triaged more quickly?

Result 5: The longer it takes to triage a bug report, the lower chance of it being marked as a true bug

Table 3 shows that the longer it takes for a report to be triaged, the more likely it will be marked as *unsure* (veracity could not be determined) and less likely marked as a *true bug*. For example, 54% of reports triaged over one year after their release dates (the “ \geq ” row) were marked as *unsure*, versus only 17% of reports triaged within one year (“<” row). Correspondingly, 11% of reports triaged over one year after their release dates were marked as *true bug*, versus 33% of those triaged within one year.

Discussion: Without a policy forcing certain bugs to be triaged, developers tend to triage the simplest bugs first:

“True, people first go after the low hanging fruits and complicated reports might stay un-triaged.” (Dev A)

Once confirmed, these quickly-triaged reports are usually easy to silence by adding a few lines of code like an extra null pointer check (the median size of a Coverity bugfix patch is 3 lines, versus 11 lines for all patches). However, these seemingly superficial bugs often indicate deeper misunderstandings of program invariants or interfaces, so the affected code should be audited more carefully. In a mailing list discussion about Coverity bugs,

given:	Pr(all reports triaged)
unconditional	46%
≥ 1 reports triaged	65%
≥ 2 reports triaged	87%

given:	Pr(all reports un-triaged)
unconditional	30%
≥ 1 reports un-triaged	55%
≥ 2 reports un-triaged	79%

Table 4: Probabilities of all reports in a file-scan session being triaged or un-triaged, for sessions with ≥ 2 reports.

one developer shows concern that others are submitting quick “fixes” rather than figuring out their root causes:

“Considering the very important flow of patches you are sending these days, I have to admit I am quite suspicious that you don’t really investigate all issues individually as you should, but merely want to fix as many bugs as possible in a short amount of time. This is not, IMVHO [in my very humble opinion], what needs to be done.” [6]

Given these natural tendencies, it might be effective to enforce policies to make developers triage more complicated but potentially critical reports and to carefully investigate each one before submitting a patch, perhaps even requiring sign-offs from multiple triagers.

If a report isn’t triaged quickly, then it might either never be triaged or be marked as *unsure*:

“Many maintainers have an inbox-is-to-do-list mentality when it comes to bugfixes. If they receive a scan report and don’t act on it quickly then it’s likely it’s left the inbox and left the maintainer’s thoughts forever.” (Dev D)

This problem of fading memories could be alleviated if reports were immediately brought to the attentions of relevant developers (e.g., those who created or recently modified the file). To do so, developers could run bug-finding tools continuously while coding (e.g., PREFast at Microsoft [11]) rather than making monolithic nightly or weekly scans over the entire codebase. Ayewah et al. suggest triaging static analysis warnings as part of the code review process [3]. Also, a bug database could periodically remind developers who are responsible for a file to look at its un-triaged bug reports.

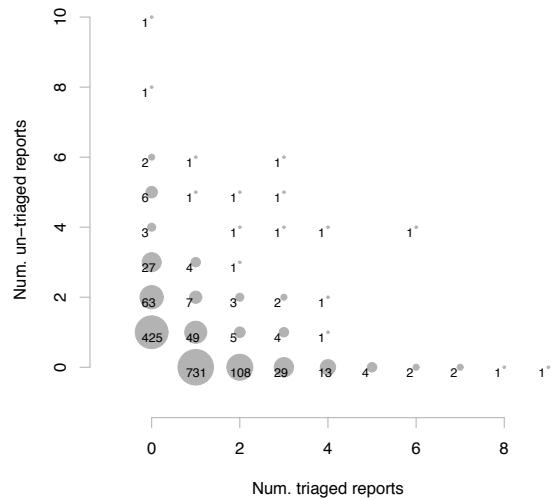


Figure 3: L-shaped clustering of triaged reports: Each dot is labeled with the number of file-scan sessions that have the given numbers of triaged and un-triaged reports.

4 Within-file clustering of triaged reports

Result 6: If one report in a file is triaged, then it’s likely that all other reports in that file will be triaged

Triaged reports are clustered in space: During a particular scan, if a developer triages one report in a file, then he/she will likely triage all other reports in that file. We call one of these sessions of triaging bugs within one file during a particular scan a *file-scan session*.

Figure 3 visualizes this clustering: Each dot represents a collection of file-scan sessions that have a given number of triaged vs. un-triaged reports. Its salient feature is the L-shaped distribution — there are many sessions along the vertical axis representing 0 triaged reports and along the horizontal axis representing 0 un-triaged reports. This pattern shows that either all reports in a session are triaged or left un-triaged. (Kremenek et al. used a similar diagram to visualize clustering of true bugs vs. false positives [10].)

Table 4 quantifies the amount of clustering: The probability that all reports in a session are triaged (or un-triaged) rise markedly when at least 1 or 2 reports are triaged (or un-triaged). The largest dots in Figure 3 are located at (1, 0) and (0, 1), representing sessions with only 1 report per file. We excluded these singleton sessions from the calculations in Table 4, since clustering is only meaningful for sessions with multiple reports.

What happened to reports in prev. scan:	Pr(triage)
0 reports triaged	50%
≥ 1 reports triaged	59%
≥ 1 marked true bug	67%
≥ 1 marked true bug and fixed	80%
≥ 1 marked false positive	56%
unconditional probability	54%

Table 5: Probabilities of reports being triaged in a file during any particular scan, given what happened to reports in that file in the scan immediately preceding it. We used the 280 files that have reports from ≥ 2 scans.

Result 7: Triaging, verifying, and fixing reports increase the probability of triaging future reports

Clustering also extends across time: If a report is triaged, then future reports in that same file are more likely to be triaged. Table 5 shows the probabilities of reports in a scan being triaged, given what happened to reports in the scan immediately preceding it. Only 50% of reports were triaged when no reports in the same file were triaged in the previous scan. If at least 1 previous report was triaged, then the conditional probability rises to 59%, and it increases further if those reports were marked as true bugs (67%) and were fixed (80%).

Each act of triaging a bug report shows that some developer cares about bugs in that file, and verifying and fixing bugs are even stronger indicators. In contrast, if developers are given the opportunity to triage a report but do not do so, then either it's too hard to diagnose or nobody cares about bugs in that file.

Result 8: False positives decrease the probability of triaging future reports

If developers mark reports in a particular scan as false positives, then they are less likely to triage future reports in the same file, versus had they marked them as true bugs (56% vs. 67% triaged):

“False positives tend to lower the maintainer’s trust of the tool and are more likely then to let future reports from the same tool slip.” (Dev D)

Looking back at Table 1, the 2 checkers with the highest false positive rates also had the largest decreases in triage rates between the initial and subsequent scans (marked with the \dagger symbol). Developers triaged most initial scan reports from those checkers (72%), but after encountering too many false positives, they triaged substantially fewer reports in subsequent scans.

However, the triage rate when previous reports were marked as false positives is still greater than when previous reports went un-triaged (56% vs. 50%), since the act

of triaging shows that somebody cares about that file.

Discussion: To encourage adoption of static analysis tools, it might be useful to assign the easiest reports (those with the highest triage rates) to developers who are new to the tool, to encourage them to keep triaging:

“The kernel is such a big project then [sic] triaging bug reports can be quite intimidating [...] Once a developer has got some confidence up in a subsystem they are more likely to step up to the plate and triage again.” (Dev D)

Also, clustering of report triaging shows that developers have sustained interest in certain files and don't simply triage reports without regard to the files they are in. Frequently-triaged files likely contain more important code. In fact, triage frequency might be a better indicator of *code importance* than number of recent patches, since we've observed that many unmaintained files still receive trivial patches when module-wide interfaces are updated.

Once we flag which files are more important to developers, we can customize bug-finding tools to perform deeper and more precise analysis on those files, which can potentially reduce false positives.

5 Static analysis bug reports as indicators of user-reported bugs

We define a *user-reported bug* as one that was not reported by Coverity or Sparse [1], the two sources that comprise the vast majority of static analysis bug reports for Linux. As a proxy, we record patches that fix user-reported bugs (rather than occurrences of such bugs) since users only report symptoms and cannot pinpoint specific files as causes; in contrast, bugfix patches and static analysis reports always target specific files.

Result 9: Files and modules with more bugs found by static analysis also contain more user-reported bugs

The Spearman's rank correlation between the number of Coverity Scan reports in each file and the number of patches that fix user-reported bugs is 0.27, which is statistically significant but somewhat weak. It's difficult to get high Spearman correlations since most files had less than 3 reports. To get a cleaner signal, Microsoft researchers used static analysis reports to predict bug density in *modules* rather than in files [11]. We also calculated correlations for bugs aggregated over entire directories (1,203 total), which serve as ad-hoc kernel modules, and our correlation grew substantially to 0.56. The Microsoft study found a similar module-level correlation of 0.58 between static analysis bugs and pre-release bugs found by QA in the Windows Server 2003 codebase [11].

Files in initial scan with:	# files	Time elapsed since initial scan on Feb 24, 2006					
		1 month	3 months	6 months	1 year	∞	entire lifetime
Percent of files containing fixes for user-reported bugs							
no Coverity reports	7,504	4%	9%	17%	35%	45%	69%
≥ 1 reports	633	13%	24%	39%	55%	66%	92%
≥ 1 triaged reports	444	14%	25%	41%	58%	68%	92%
≥ 2 reports	197	17%	28%	45%	65%	75%	96%
Mean number of fixes for user-reported bugs per file							
no Coverity reports	7,504	0.06	0.12	0.27	0.61	0.98	2.8
≥ 1 reports	633	0.17	0.38	0.72	1.35	2.17	7.4
≥ 1 triaged reports	444	0.18	0.40	0.75	1.44	2.32	7.8
≥ 2 reports	197	0.28	0.63	1.06	1.86	2.79	9.4

Table 6: Numbers of initial scan Coverity reports versus numbers of future fixes for user-reported bugs, calculated for all 8,137 .c files alive during the initial scan. Values don’t change considerably for ≥ 3 reports.

Surprisingly, reports that developers have marked as false positives still somewhat correlate with user-reported bugs, with a file-level correlation of 0.15 and directory-level correlation of 0.42. One possible explanation is that static analysis tools are more likely to produce false positives when analyzing more semantically-complex and convoluted code, which is more likely to contain latent functional correctness bugs that users will later report. For example, a veteran developer triaged a static buffer overrun report in an InfiniBand networking driver, marked it as a true bug, and then a day later remarked it as a false positive, noting in the bug database:

“It’s horrible, but after looking deeper, including looking at the callers, I’m now convinced it’s correct (this code only gets called in 64bit kernels where longs are double the size of ints).” (Dev A)

Files like this one with code that even baffles a veteran developer probably also contain subtle correctness bugs.

Result 10: Bugs found by static analysis can predict future user-reported bugs in the same file

Not only are numbers of Coverity and user-reported bugs correlated, but the presence of Coverity bugs can foreshadow a file having user-reported bugs in the future.

We considered all 8,137 .c files alive during the initial scan on Feb 24, 2006, to simplify calculations and to prevent biases due to files being added over time. We partitioned files into subsets based on how many reports from that initial scan affected each file. For example, the “ ≥ 1 reports” rows of Table 6 are for all files with at least 1 report. For each file, we counted the number of bugfix patches for user-reported bugs in the subsequent 1 month, 3 months, 6 months, 1 year, and the rest of the file’s life (the “ ∞ ” column). We also counted bugfix

patches over each file’s *entire lifetime*, which takes into account patches that occurred before the initial scan.

As a sanity check, the numbers increase across each row of Table 6 because the more time elapses, the more likely it is for files to receive bugfix patches.

Scanning down each column, we can compare values across files with varying numbers of Coverity reports. More reports boosts the chances of future (fixes for) user-reported bugs, as shown by the numbers increasing *down each column*. Note that having at least one triaged report is a slightly better predictor than simply having one report, because triaging shows that someone is actively monitoring that file. For instance, the “1 month” column shows that 13% of files with initial scan reports had fixes for user-reported bugs in the next month, versus only 4% of files with no reports (over 3X greater). The mean number of user-reported bugs per file — 0.17 vs. 0.06 — was also 3X greater. This 3X increase is consistent across all time scales.

Discussion: Static analysis tools excel at finding generic errors (e.g., like those in Table 1) but cannot usually find higher-level functional correctness bugs like those that users report (e.g., *driver X doesn’t do the right thing when fed this input*). However, results like ours and related work on a commercial codebase at Microsoft [11] show that static analysis tools can be useful for pointing developers towards regions within the codebase that are more error-prone, which is cost-effective because these tools can be run automatically and continuously.

In fact, some kernel developers advocate using static analysis tools in exactly this manner: directing developers’ attentions towards potentially buggy code:

“Coverity and similar tools are a true opportunity for us to find out and study suspect parts of our code. Please do not misuse these tools! The goal is NOT to make the tools happy next

time you run them, but to actually fix the problems, once and for all. If you focus too much on fixing the problems quickly rather than fixing them cleanly, then we forever lose the opportunity to clean our code, because the problems will then be hidden.” [6]

This use case could partially explain the low incidence of fixes (only 8% of triaged reports were confirmed as bugs and fixed). Developers might want to purposely leave in errors as markers for “suspect parts” of the codebase until that code can be properly audited and fixed.

6 Related Work

To our knowledge, Google researchers did the closest related work in terms of studying developer responses to static analysis bug reports. Ayewah et al. described experiences with deploying FindBugs at Google [3], where two dedicated test engineers triaged all bug reports. In contrast, our study focuses on open source code where 26 kernel developers triaged reports. Ruthruff et al. built a logistic regression model to predict which FindBugs reports at Google were likely to be triaged or marked as false positives [13], using factors similar to those we describe in Section 2 and in our technical report [8].

Nagappan and Ball found a correlation between bugs reported by the PREfix/PREFast static analysis tools and pre-release defects found by testers within modules in Microsoft Windows Server 2003 [11]. We performed a similar analysis in Section 5 and found similar correlations, albeit using a different analysis tool and codebase.

In terms of static analysis bug reports for the Linux kernel, Chou et al. quantified distributions and lifetimes of kernel bugs found by a precursor of Coverity Prevent [4]. Kremeneck et al. proposed a technique for incorporating developer feedback to filter and rank reports so as not to overwhelm triagers, and performed an evaluation on bug reports issued for kernel code [10].

Other work related to bug report triaging include prioritization and ranking of reports [9], optimizing assignments of triagers to specific reports [2], and graphical user interfaces for facilitating the triaging process [12].

7 Limitations

We evaluated developer responses to static analysis bug reports in an open source setting where there were no organizational policies for triaging or fixing these bugs. Findings might differ in a corporate setting where static analysis is integrated into the workflow. With any empirical study, we must be cautious about over-generalizing based solely upon data analysis; trying to infer human intentions from code-related artifacts is a difficult problem.

Thus, we tried to support our claims using anecdotes gathered from kernel developers. Also, similar findings from other researchers working with different tools and codebases make our results more generalizable.

Acknowledgments

We thank David Maxwell for providing the Coverity dataset, Greg Little and Derek Rayside for help with questionnaire design, kernel developers who responded to our questionnaire, Joel Brandt, Cristian Cadar, Imran Haque, David Maxwell, Derek Rayside, and our shepherd George Candea for comments on this paper and its earlier drafts. This research was supported by NSF TRUST grant CCF-0424422 and the NDSEG fellowship.

References

- [1] Sparse – A Semantic Parser for C, <http://www.kernel.org/pub/software-devel/sparse/>.
- [2] ANVIK, J., HIEW, L., AND MURPHY, G. C. Who should fix this bug? In *ICSE ’06: Proceedings of the 28th international conference on Software engineering* (May 2006), pp. 361–370.
- [3] AYEWAH, N., HOVEMEYER, D., MORGENTHALER, J. D., PENIX, J., AND PUGH, W. Using static analysis to find bugs. *IEEE Softw.* 25, 5 (2008), 22–29.
- [4] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *SOSP ’01: Proceedings of the symposium on Operating Systems Principles* (2001), pp. 73–88.
- [5] COVERITY. Coverity Scan Open Source Report 2008, <http://scan.coverity.com/report/>.
- [6] DELVARE, J. Email: Re: Do not misuse Coverity please — <http://lkml.org/lkml/2005/3/27/131>. *Linux Kernel Mailing List* (Mar. 2005).
- [7] GUO, P. J. Email: research questionnaire about kernel development — <http://lkml.org/lkml/2008/8/7/98>. *Linux Kernel Mailing List* (Aug. 2008).
- [8] GUO, P. J. Using logistic regression to predict developer responses to Coverity Scan bug reports. Tech. Rep. CSTR 2008-04, Stanford Computer Systems Lab, Stanford, CA, July 2008.
- [9] KIM, S., AND ERNST, M. D. Which warnings should I fix first? In *ESEC-FSE ’07: Proceedings of symposium on the foundations of software engineering* (2007), ACM, pp. 45–54.
- [10] KREMENEK, T., ASHCRAFT, K., YANG, J., AND ENGLER, D. Correlation exploitation in error ranking. *SIGSOFT Softw. Eng. Notes* 29, 6 (2004), 83–93.
- [11] NAGAPPAN, N., AND BALL, T. Static analysis tools as early indicators of pre-release defect density. In *ICSE ’05: Proceedings of the 27th international conference on Software engineering* (2005), ACM, pp. 580–586.
- [12] PHANG, K. Y., FOSTER, J. S., HICKS, M., AND SAZAWALI, V. Path projection for user-centered static analysis tools. In *PASTE ’08: Proceedings of the 8th ACM workshop on Program analysis for software tools and engineering* (2008).
- [13] RUTHRUFF, J. R., PENIX, J., MORGENTHALER, J. D., ELBAUM, S., AND ROTHERMEL, G. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE ’08: Proceedings of the 30th international conference on Software engineering* (2008), ACM, pp. 341–350.

Hardware Execution Throttling for Multi-core Resource Management *

Xiao Zhang Sandhya Dwarkadas Kai Shen

Department of Computer Science, University of Rochester

{xiao, sandhya, kshen}@cs.rochester.edu

Abstract

Modern processors provide mechanisms (such as duty-cycle modulation and cache prefetcher adjustment) to control the execution speed or resource usage efficiency of an application. Although these mechanisms were originally designed for other purposes, we argue in this paper that they can be an effective tool to support fair use of shared on-chip resources on multi-cores. Compared to existing approaches to achieve fairness (such as page coloring and CPU scheduling quantum adjustment), the execution throttling mechanisms have the advantage of providing fine-grained control with little software system change or undesirable side effect. Additionally, although execution throttling slows down some of the running applications, it does not yield any loss of overall system efficiency as long as the bottleneck resources are fully utilized. We conducted experiments with several sequential and server benchmarks. Results indicate high fairness with almost no efficiency degradation achieved by a hybrid of two execution throttling mechanisms.

1 Introduction

Modern multi-core processors may suffer from poor fairness with respect to utilizing shared on-chip resources (including the last-level on-chip cache space and the memory bandwidth). In particular, recent research efforts have shown that uncontrolled on-chip resource sharing can lead to large performance variations among co-running applications [5, 17]. Such poor performance isolation makes an application’s performance hard to predict and consequently it hurts the system’s ability to provide quality-of-service support. Even worse, malicious applications can take advantage of such obliviousness to on-chip resource sharing to launch denial-of-service attacks and starve other applications [10].

Much research has tried to tackle the issue of fair resource utilization on multi-core processors. Some require significant new hardware mechanisms that are not available on commodity platforms [1, 3, 14, 17]. Without extra hardware support, the operating system must resort

to software techniques such as page coloring to achieve cache partitioning [4, 9, 13, 15, 16] and CPU scheduling quantum adjustment to achieve fair resource utilization [5]. However, page coloring requires significant changes in the operating system memory management, places artificial constraints on system memory allocation policies, and incurs expensive re-coloring (page copying) costs in dynamic execution environments. CPU scheduling quantum adjustment suffers from its inability to provide fine-grained quality of service guarantees.

In this paper, we argue that hardware execution throttling can efficiently manage on-chip shared resources with much less complexity and overhead than existing alternatives, while providing the necessary granularity of quality of service. Specifically, we investigate the use of existing hardware mechanisms to control the cache/bandwidth consumption of a multi-core processor. Commodity processors are deployed with mechanisms (e.g., duty cycle modulation and dynamic voltage and frequency scaling) to artificially slow down execution speed for power/thermal management [7]. By throttling down the execution speed of some of the cores, we can control an application’s relative resource utilization to achieve desired fairness or other quality-of-service objectives. In addition to direct throttling of CPU speed, we also explore the existing mechanism of controlling L1 and L2 cache hardware prefetchers. Different cache prefetching configurations also allow us to manage an application’s relative utilization of the shared memory bandwidth and cache space.

2 Multi-core Resource Management Mechanisms

2.1 Hardware Execution Throttling

One mechanism to throttle a CPU’s execution speed available in today’s multi-core platforms is dynamic voltage and frequency scaling. However, on some multi-core platforms, sibling cores often need to operate at the same frequency [11]. Intel provides another mechanism to throttle per-core execution speed, namely, *duty-cycle modulation* [7]. Specifically, the operating system can specify a portion (e.g., multiplier of 1/8) of regular CPU cycles as duty cycles by writing to the logical processor’s IA32_CLOCK_MODULATION register. The processor is effectively halted during non-duty cycles. Duty-cycle

*This work was supported in part by the NSF grants CNS-0411127, CAREER Award CCF-0448413, CNS-0509270, CNS-0615045, CNS-0615139, CCF-0621472, CCF-0702505, and CNS-0834451; by NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01; and by several IBM Faculty Partnership Awards.

Prefetchers	Description
L1 IP	Keeps track of instruction pointer and looks for sequential load history.
L1 DCU	When detecting multiple loads from the same line within a time limit, prefetches the next line.
L2 Adjacent Line	Prefetches the adjacent line of required data.
L2 Stream	Looks at streams of data for regular patterns.

Table 1: Brief description of four L1/L2 cache prefetchers on Intel Core 2 Duo processors [7].

modulation was originally designed for thermal management and was also used to simulate an asymmetric CMP in recent work [2].

Execution throttling is not work-conserving since it leaves resources partially idle while there are still active tasks. Consequently, there is potential cause for concern about lost efficiency in the pursuit of fairness. We argue that careful execution throttling only affects the relative resource use among co-running applications. It should not degrade the overall system efficiency as long as the bottleneck resource (shared cache space or memory bandwidth) is fully utilized.

Today’s commodity processors often perform *hardware prefetching*, which helps hide memory latency by taking advantage of bandwidth not being used by on-demand misses. However, in a multi-core environment, the result might be contention with the on-demand misses of concurrently executing threads. The hardware prefetchers are usually configurable. For example, on Intel Core 2 Duo processors, there are two L1 cache prefetchers (DCU and IP prefetchers) and two L2 cache prefetchers (adjacent line and stream prefetchers) [7]. Table 1 briefly describes the prefetchers on our test platform. Each can be selectively turned on/off, providing partial control over a thread’s bandwidth utilization.

Both duty-cycle modulation and prefetcher adjustment can be used to throttle an application’s execution speed. The former directly controls the number of accesses to the cache (the execution speed), thereby affecting cache pressure and indirectly the bandwidth usage, while the latter directly controls bandwidth usage, thereby affecting cache pressure and indirectly affecting execution speed. Adjusting the duty cycle alone might not result in sufficient bandwidth reduction if the prefetching is aggressive, while adjusting the prefetching alone might not reduce cache pressure sufficiently. Both mechanisms can be combined to arrive at fair resource allocation.

On our platform, configuring the duty cycle takes $265 + 350$ (read plus write register) cycles; configuring the prefetchers takes $298 + 2065$ (read plus write register) cycles. The control registers also specify other features in addition to our control targets, so we need to read their values before writing. The longer time for a new prefetching configuration to take effect is possibly due to clearing obsolete prefetch requests in queues. Roughly speaking, the costs of configuring duty cycle modula-

tion and cache prefetcher are 0.2 and 0.8 microseconds respectively on our 3.0 GHz machine.

Enabling these mechanisms requires very little operating system software modification. Our changes to the Linux kernel source are ~ 40 lines of code in a single file.

2.2 Alternative Mechanisms

Cache Partitioning Page coloring, a technique originally proposed for cache conflict mitigation [8, 12], is a software technique that manipulates mapping between memory and cache. Memory pages that are mapped to the same cache blocks are labeled to be in the same color. By manipulating the allocation of colors to applications, the operating system can partition a cache at page granularity (strictly speaking, at a granularity of *PageSize* times *CacheAssociativity*). The maximum number of colors that a platform can support is determined by $\frac{\text{CacheSize}}{\text{PageSize} \times \text{CacheAssociativity}}$.

Page coloring has recently been used to manage cache allocation [4, 9, 13, 15] by isolating cache space usage among applications. However, page coloring has a number of important drawbacks [16]. First, during dynamic executions in multi-programmed environments, the resource manager may decide to change an application’s cache share due to a priority change or a change in the set of simultaneously executing processes. This would require re-coloring of a potentially large number of memory pages with each re-coloring typically requiring an expensive page copy. As a quantitative reference, copying a single page costs around 3 microseconds on our platform, which is already much more expensive than the configuration (or re-configuration) of hardware execution throttling mentioned earlier.

The second drawback is that page coloring enforces strict memory to cache mapping and introduces artificial memory allocation constraints. For example, an application allocated one eighth of all cache colors is also entitled to only one eighth of the total memory space. This artificial memory allocation constraint may force an application to run out of its entitled memory space while many free pages are still available in other colors.

Finally, compared to hardware execution throttling, page coloring requires more significant changes in the operating system memory management code. Our incomplete implementation of page coloring (without full support for page re-coloring) involves more than 700 lines of Linux source code changes in 10 files.

In addition to software-based cache management mechanisms, several hardware-level mechanisms have also been proposed [1, 14, 17]. They generally require adding new hardware counters/tags to monitor fine-grained cache usage, and modify the cache replacement policy based on applications’ resource entitlements. It is also possible to implement associativity-based cache par-

titioning (called column caching in [3]), which is a trade-off between control flexibility and deployment overhead. While such hardware mechanisms could be beneficial, we focus here on mechanisms available in today's commodity platforms.

CPU Scheduling Quantum Adjustment Fedorova *et al.* proposed a software method to maintain fair resource usage on multi-cores [5]. They advocate adjusting the CPU scheduling time quantum to increase or decrease an application's relative CPU share. By compensating/penalizing applications under/over fair cache usage, the system tries to maintain equal cache miss rates across all applications (which is considered fair). To derive the fair cache miss rate, they profile an application's behavior with several different co-runners.

The key drawback of CPU scheduling quantum adjustment is that it only achieves fairness at granularities comparable to the scheduling time quantum. This would lead to unstable performance of fine-grained tasks (such as individual requests in a server system).

3 Evaluation and Results Analysis

We enabled the duty-cycle modulation and cache prefetcher adjustment mechanisms by modifying the Linux 2.6.18 kernel. Our experiments were conducted on an Intel Xeon 5160 3.0 GHz "Woodcrest" dual-core platform. The two cores share a single 4 MB L2 cache (16-way set-associative, 64-byte cache line, 14 cycle latency, writeback).

Our evaluation benchmarks include three programs from SPECCPU2000: swim, mcf, and equake. We also employ two server-style benchmarks (SPECjbb2005 and SPECweb99) in our evaluation. SPECjbb is configured with four warehouses and a 500 MB heap size. SPECweb is hosted on the Apache web server 1.3.33. When running alone, swim, mcf, and equake take 136.1, 46.1, and 67.5 seconds respectively to complete. We bind each server application to a single core to get its baseline performance. SPECjbb delivers a throughput of 17794.4 operations/second and SPECweb delivers a throughput of 361.5 web requests/second.

Optimization Goal and Policy Settings We measure several approaches' ability to achieve fairness and, in addition, evaluate their efficiency. There are several possible definitions of fair use of shared resources [6]. The particular choice of fairness measure should not affect the main purpose of our evaluation. In our evaluation, we use *communist fairness*, or equal performance degradation compared to a standalone run for the application. Based on this fairness goal, we define an *unfairness factor* metric as the coefficient of variation (standard deviation divided by the mean) of all applications' performance normalized to that of their individual standalone

run. We also define an *overall system efficiency* metric as the geometric mean of all applications' normalized performance.

We consider two execution throttling approaches. One is based on the per-core duty cycle modulation. Another is a hybrid approach that employs both duty cycle modulation and cache prefetcher adjustment. We implement two additional approaches in the Linux kernel for the purpose of comparison: an ideal page coloring approach (one that uses a statically defined cache partition point and incurs no page recoloring cost) and scheduling quantum adjustment using an idle process to control the amount of CPU time allocated to the application process. As a base for comparison, we also consider *default sharing*—running two applications on a dual-core processor under the default hardware/software resource management.

For each approach other than default sharing, we manually try all possible policy decisions (*i.e.*, page coloring partitioning point, duty cycle modulation ratio, cache prefetcher configuration, and idle process running time) and report the result for the policy decision yielding the best fairness. Since the parameter search space when combining duty cycle modulation and prefetcher configuration is large, we explore it in a genetic fashion. Specifically, we first select default and a few duty cycle modulation settings that achieve reasonably good fairness and then tune their prefetchers to find a best configuration. In most cases, duty-cycle modulation and duty-cycle & prefetch reach the same duty-cycle ratio except for {swim, SPECjbb}. In this case, setting swim's duty-cycle to 5/8 has a similar throttling effect to disabling its L2 stream prefetcher.

Figure 1 illustrates the page coloring-based cache partition settings yielding the best fairness. Table 2 lists the best-fairness policy settings for the hybrid hardware throttling (duty cycle modulation and cache prefetcher configuration) and scheduling quantum adjustment respectively. All cache prefetchers on our platform are per-core configurable except the L2 adjacent line prefetcher.

Best Fairness Figure 2 shows the fairness results (in terms of the unfairness factor) when running each possible application pair on the two cores (running two instances of the same application shows an unfairness factor close to 0 in all cases, so we do not present these results in the figure). On average, the unfairness factor is 0.191, 0.028, 0.025, 0.027, and 0.017 for default sharing, page coloring, scheduling quantum adjustment, duty-cycle modulation, and duty-cycle & prefetch, respectively. Default sharing demonstrates a higher unfairness factor in several cases. The level of unfairness is a function of the properties of the co-running applications. If their cache and bandwidth usage requirements are similar, the unfairness factor is low. If the requirements are

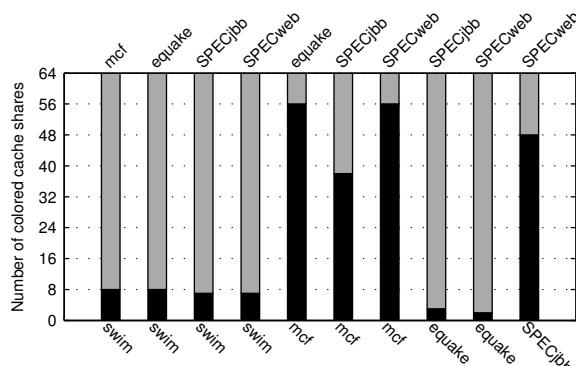


Figure 1: Cache partition settings under page coloring to achieve best fairness. Our experimental platform supports at most 64 colors and therefore the shared 4 MB L2 cache is divided into 64 shares. Results are shown for all possible application pairs from our five benchmarks (pairing an application with itself results in the cache being partitioned in half).

significantly different, and if the sum of the requirements exceeds the available resource, the unfairness factor is high due to uncontrolled usage.

Ideal page coloring-based cache partitioning also shows some variation in the unfairness factor across benchmark pairs. In particular, {swim, SPECweb} shows a comparatively higher unfairness factor due to two competing effects. Under page coloring, if swim was entitled to a very small portion of the cache space, its mapped memory pages might be less than its required memory footprint, resulting in thrashing (page swapping to disk). If swim’s cache usage is not curtailed, SPECweb’s normalized performance is significantly affected. These competing constraints result in page coloring not achieving good fairness (overall efficiency is also lower than with default sharing) in this case.

While both page coloring and execution throttling achieve better fairness than default sharing, the combination of duty cycle modulation and prefetching control achieves a uniformly low unfairness factor below 0.03. This uniform fairness is achieved without the additional (not accounted for; we elaborate further later in this section) overheads of page coloring. One can extend these fairness goals to additional management objectives like proportional resource allocation.

The scheduling quantum adjustment obtains similar low unfairness factor to hardware throttling. However, these results are calculated based on coarse-grained performance measurement (*i.e.*, at the scale of whole application execution). When examined at finer granularity, performance fluctuates (see Figure 4; we elaborate further later in this section), suggesting unstable fairness.

Efficiency At Best Fairness Figure 3 shows evaluation results on the overall system efficiency (when the best

Co-running applications	Duty-cycle modulation	Hardware throttling Non-default cache prefetcher setup	Scheduling quantum adjustment
swim mcf	5/8 Default	Default Default	100/30 NA
swim equake	7/8 Default	Default Enable L1 DCU	100/20 NA
swim SPECjbb	Default Default	Disable L2 stream Default	100/40 NA
swim SPECweb	6/8 Default	Default Default	100/30 NA
mcf equake	Default 6/8	Disable L2 adjacent line Disable L2 adjacent line	NA 100/25
mcf SPECjbb	Default Default	Default Default	NA NA
mcf SPECweb	Default Default	Disable L2 adj. & stream Disable L2 adjacent line	100/5 NA
equake SPECjbb	6/8 Default	Enable L1 DCU Enable L1 DCU	100/30 NA
equake SPECweb	7/8 Default	Default Default	100/30 NA
SPECjbb SPECweb	Default Default	Disable L2 stream Default	NA NA

Table 2: Configurations of hardware throttling and scheduling quantum adjustment to achieve best fairness. The duty-cycle modulation must be a multiplier of 1/8 on our platform. The default hardware throttling configuration is full execution speed (or 8/8), plus enabled L1 IP, disabled L1 DCU, enabled L2 adjacent line, and enabled L2 stream prefetchers. The scheduling quantum adjustment adds an idle process to squeeze one’s CPU share. For example, “100/30” means every round, the application and idle process alternate, running for 100 and 30 milliseconds, respectively. “NA” implies no idle process was used. Results are shown for all possible application pairs from our five benchmarks (pairing an application with itself results in the use of default configurations).

fairness is achieved under each approach). Here we also include the efficiency results of running two identical applications. Note that fairness can be trivially achieved for these cases by all mechanisms (*i.e.*, equal cache partitioning under page coloring, equal setups on both cores for execution throttling and prefetching, no scheduling quantum adjustment). However, as the results demonstrate, system efficiency for some of the approaches varies. Note that for the prefetcher adjustment, we may choose a (identical) non-default prefetcher setting for efficiency gain. Specifically, we do so in two instances: for {mcf, mcf}, we enable the L1 DCU prefetcher and disable the L2 adjacent line prefetcher; for {SPECjbb, SPECjbb}, we enable the L1 DCU prefetcher and disable the L2 stream prefetcher. On average, the efficiency of all approaches is similar (roughly 0.65). Specific cases where significant differences occur are discussed below.

For {mcf, equake} and {equake, SPECjbb}, equake aggressively accesses the L2 cache and makes its co-runner suffer intensive cache conflicts. Our miss ratio profile shows that equake is not cache space sensitive, demonstrating only a 2% increase in cache miss ratio when varying the L2 cache space from 4 MB to 512 KB.

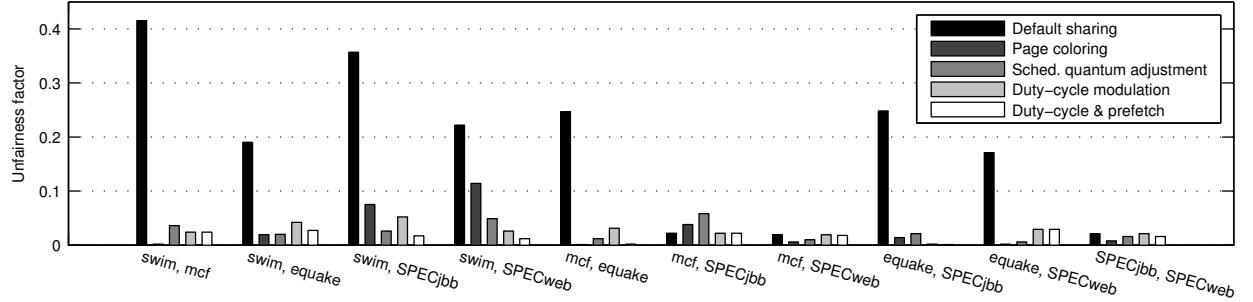


Figure 2: Comparison of the unfairness factor (the lower the better).

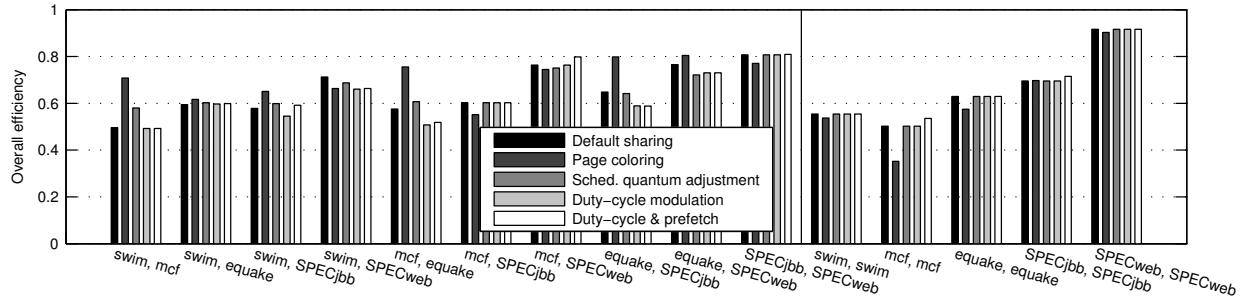


Figure 3: Comparison of the overall system efficiency when each approach is optimized for best fairness. We also include results of running two identical applications.

By constraining equake to a small portion of the L2 cache, page coloring can effectively prevent pollution of the co-runner’s cached data without hurting equake’s performance. Hardware throttling approaches do not fundamentally solve inter-application cache conflicts and need to slow down equake’s execution dramatically to achieve “fair” cache sharing. In these cases, hardware throttling has roughly 10% efficiency degradation while page coloring improves efficiency by 23~30% relative to default sharing. The scheduling quantum adjustment also achieves better efficiency than hardware throttling in these two cases. This is because equake is less vulnerable to inter-application cache conflicts than the other application. By introducing an idle process to reduce equake’s co-running time with the other application, it greatly mitigates the negative cache conflict impact on the other application and therefore boosts overall efficiency. Similar analysis also applies to {swim, mcf}.

For {mcf, mcf}, page coloring shows about 30% degraded efficiency relative to default sharing. mcf is a cache-space-sensitive application. Under page coloring, each instance of mcf gets half the cache space. When it runs alone, mcf has a 17% cache miss ratio when given 4 MB L2 cache and that number increases to 35% with a 2 MB L2 cache. Under sharing, two mcf’s data accesses are well interleaved such that each gets better performance than that using a 2 MB cache. Since the two instances are equally aggressive in requesting cache re-

sources with default sharing, the unfairness factor remains low. By tuning the prefetching, hardware throttling can improve efficiency by 6% over the default.

Costs of Dynamic Page Re-Coloring The high efficiency of page coloring is obtained assuming a somewhat ideal page coloring mechanism, meaning that the cache partition point is statically determined and no page recoloring is needed. In reality, a dynamic environment would involve adjusting cache colors and partition points based on changes in the execution environment. Without extra hardware support, re-coloring a page means copying a memory page and it usually takes several micro-seconds on typical commodity platforms (3 microseconds on our test platform). Assuming an application must re-color half of its working set every scheduling quantum (default 100 milliseconds in Linux), our five benchmarks would incur 18–180% slowdown due to page re-coloring (the smallest working set is around 50 MB (quake) and the largest around 500 MB+ (SPECjbb)). This would more than negate any efficiency gain by page coloring.

Instability of Scheduling Quantum Adjustment While scheduling quantum adjustment achieves fairness at coarse granularities comparable to the scheduling quantum size, it may cause fluctuating performance for fine-grained tasks such as individual requests in a server system. As a demonstration, we run SPECjbb and swim on a dual-core chip. Consider a hypothetical resource management scenario where we need to

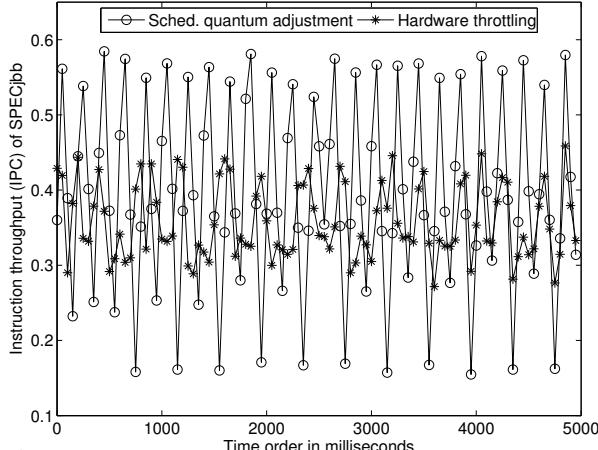


Figure 4: SPECjbb’s performance when its co-runner swim is regulated using two different approaches: scheduling quantum adjustment (default 100-millisecond quantum) and hardware throttling. Each point in the plot represents performance measured over a 50-millisecond window.

slow down swim by a factor of two. We compare two approaches—the first adds an equal-priority idle process on swim’s core; the second throttles the duty cycle at swim’s core to half the full speed. Figure 4 illustrates SPECjbb’s performance over time under these two approaches. For scheduling quantum adjustment, SPECjbb’s performance fluctuates dramatically because it highly depends on whether its co-runner is the idle process or swim. In comparison, hardware throttling leads to more stable performance behaviors due to its fine-grained execution regulation.

4 Conclusion

This paper investigates the use of hardware-assisted execution throttling (duty cycle modulation combined with L1/L2 cache prefetcher configuration) for regulating fairness in modern multi-core processors. We compare against page coloring-based cache partitioning and scheduling time quantum adjustment. Our results demonstrate that simple hardware-assisted techniques to throttle an application’s execution speed can achieve high fairness at fine granularity without the drawbacks of page re-coloring costs.

In this work, we have focused on demonstrating the relative benefits of the various resource control mechanisms. Built on a good mechanism, it may still be challenging to identify the best control policy during online execution and exhaustive search of all possible control policies may be very expensive. In such cases, our hardware execution throttling approaches are far more appealing than page coloring due to our substantially cheaper re-configuration costs. Nevertheless, more efficient techniques to identify the best control policy are desirable. In future work, we plan to explore

feedback-driven policy control via continuous tracking of low-level performance counters such as cache miss ratio and instructions per cycle executed, in addition to application-level metrics of execution progress.

References

- [1] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *15th Int’l Symp. on High-Performance Computer Architecture*, Raleigh, NC, Feb. 2009.
- [2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Int’l Symp. on Computer Architecture*, pages 506–517, 2005.
- [3] D. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, MIT, 1999.
- [4] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th Int’l Symp. on Microarchitecture*, pages 455–468, Orlando, FL, Dec. 2006.
- [5] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th Int’l Conf. on Parallel Architecture and Compilation Techniques*, pages 25–36, Brasov, Romania, Sept. 2007.
- [6] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource. In *15th Int’l Conf. on Parallel Architectures and Compilation Techniques*, pages 13–22, Seattle, WA, 2006.
- [7] IA-32 Intel architecture software developer’s manual, 2008.
- [8] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. on Computer Systems*, 10(4):338–359, Nov. 1992.
- [9] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Int’l Symp. on High-Performance Computer Architecture*, Salt Lake, UT, Feb. 2008.
- [10] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security Symp.*, pages 257–274, Boston, MA, 2007.
- [11] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and thermal management in the Intel Core Duo processor. *Intel Technology Journal*, 10(2):109–122, 2006.
- [12] T. Romer, D. Lee, B. Bershad, and J. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *First USENIX Symp. on Operating Systems Design and Implementation*, pages 255–266, Monterey, CA, Nov. 1994.
- [13] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *41st Int’l Symp. on Microarchitecture*, Lake Como, ITALY, Nov. 2008.
- [14] G. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *IASTED Int’l Conf. on Parallel and Distributed Computing and Systems*, Anaheim, CA, Aug. 2001.
- [15] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, San Diego, CA, June 2007.
- [16] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *4th European Conf. on Computer systems*, Nuremberg, Germany, Apr. 2009.
- [17] L. Zhao, R. Iyer, R. Illikkal, J. Moses, D. Newell, and S. Makineni. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *16th Int’l Conf. on Parallel Architecture and Compilation Techniques*, pages 339–352, Brasov, Romania, Sept. 2007.

Reducing Seek Overhead with Application-Directed Prefetching

Steve VanDeBogart

Christopher Frost

Eddie Kohler

UCLA

<http://libprefetch.cs.ucla.edu/>

Abstract

An analysis of performance characteristics of modern disks finds that prefetching can improve the performance of nonsequential read access patterns by an order of magnitude or more, far more than demonstrated by prior work. Using this analysis, we design prefetching algorithms that make effective use of primary memory, and can sometimes gain additional speedups by reading unneeded data. We show when additional prefetching memory is most critical for performance. A contention controller automatically adjusts prefetching memory usage, preserving the benefits of prefetching while sharing available memory with other applications. When implemented in a library with some kernel changes, our prefetching system improves performance for some workloads of the GIMP image manipulation program and the SQLite database by factors of 4.9x to 20x.

1 Introduction

Modern magnetic disks are, as is well known, dramatically slower at random reads than sequential reads. Technological progress has exacerbated the problem; disk throughput has increased by a factor of 60 to 85 over the past twenty-five years, but seek times have decreased by a factor of only 15.¹ Disks are less and less like random-access devices in terms of performance. Although flash memory reduces the cost differential of random accesses (at least for reads—many current SSD disks have terrible performance on small random writes), disks continue to offer vast amounts of inexpensive storage. For the foreseeable future, it will remain important to optimize the performance of applications that access disk-like devices—that is, devices with much faster sequential than random access.

Operating systems heavily optimize their use of disks, minimizing the volume of transferred data while opportunistically striving to make requests sequential. Large buffer caches ensure that disk reads are only done when necessary, write buffering helps to batch and minimize writes, disk scheduling reorders disk requests to group them and minimize seek distances, and readahead expands small read requests into more efficient large requests by predicting applications’ future behavior. These techniques apply well to workloads whose accesses are already sequential or near-sequential, for which they

achieve performance near hardware capabilities. For nonsequential access patterns, however, the techniques break down. Readahead implementations, for example, often turn off after detecting such patterns: future nonsequential accesses are by nature hard to predict, making it more likely that prediction mistakes would pollute the buffer cache with irrelevant data. Unfortunately, though careful design of application on-disk data structures can make common-case accesses sequential, many applications must sometimes access data nonsequentially—for instance, to traverse a giant database by a non-primary index—and any nonsequential access pattern is radically slow.

The best solution to this performance problem is to avoid critical-path disk access altogether, such as by obtaining enough memory to hold all application data. Failing that, distributing data over several disks or machines can reduce the overall cost of random access by performing seeks in parallel [18]. However, these solutions may not always apply—even resource-constrained users can have large data sets—and any technique that speeds up single-disk nonsequential accesses is likely to improve the performance of distributed solutions.

We present an *application-directed prefetching* system that speeds up application performance on single-disk nonsequential reads by, in some cases, more than an order of magnitude.

In application-directed prefetching systems, the application informs the storage system of its intended upcoming reads. (Databases, scientific workloads, and others are easily able to calculate future accesses [13, 17].) Previous work on application-directed caching and prefetching demonstrated relatively low speedups for single-process, single-disk workloads (average speedup 26%, maximum 49%) [2, 18]. However, this work aimed to overlap CPU time and I/O fetch time without greatly increasing memory use, and thus prefetched relatively little data from disk (16 blocks) just before a process needed it. Our system, libprefetch, aims solely to minimize I/O fetch time, a better choice given today’s widened gap between processor and disk performance. The prefetching system is aggressive, fetching as much data as fits in available memory. It is also relatively simple, fitting in well with existing operating system techniques; most code is in a user-space library. Small,

but critical, changes in kernel behavior help ensure that prefetched data is kept until it is used. A contention controller detects changes in available memory and compensates by resizing the prefetching window, avoiding performance collapse when prefetching applications compete for memory and increasing performance when more memory is available. Our measurements show substantial speedups on test workloads, such as a 20x speedup on a SQLite table scan of a data set that is twice the size of memory. Running concurrent instances of applications with libprefetch shows similar factors of improvement.

Our contributions include our motivating analysis of seek time; our prefetching algorithm; the libprefetch interface, which simplifies applications’ access to prefetching; the contention controller that prevents libprefetch from monopolizing memory; and our evaluation. Section 2 describes related work, after which Section 3 uses disk benchmarks to systematically build up our prefetching algorithm. Section 4 describes the libprefetch interface and its implementation. Finally, Sections 5 and 6 evaluate libprefetch’s performance and conclude.

2 Related Work

Fueled by the long-growing performance gulf between disk and CPU speeds, considerable research effort has been invested in improving disk read performance by caching and prefetching. Prefetching work in particular has been based on predicted, application-directed, and inferred disk access patterns.

Disk Modeling Ruemmler and Wilkes [19] is the classic paper on disk performance modeling. Our seek time observations complement those of Schlosser et al. [20]; like them, we use our observations to construct more effective ways to use a disk.

Predicting Accesses Operating systems have long employed predictive readahead algorithms to speed up sequential file access. This improves performance for many workloads, but can retard performance if future accesses are mispredicted. As a result, readahead algorithms usually don’t try to improve less predictable access patterns, such as sequential reads of many small files or nonsequential reads of large files.

Dynamic history-based approaches [5, 8, 12, 14–16, 23, 27] infer access patterns from historical analysis, and so are not limited to simple patterns. However, requiring historical knowledge has limitations: performance is not improved until a sufficient learning period has elapsed, non-repetitive accesses are not improved at all, and the historical analysis can impose significant memory and processing overheads.

Application-Directed Accesses Cao et al. [2] and Patterson et al. [18] present systems like libprefetch where

applications convey their access patterns to the file system to increase disk read performance. While all three systems prefetch data to reduce application runtime, the past decade’s hardware progress has changed the basic disk performance bottlenecks. Whereas prior systems prefetch to hide disk latency by overlapping CPU time and I/O time, libprefetch prefetches data to minimize I/O time and increase disk throughput by reducing seek distances. This approach permits much greater performance increases on today’s computers. Specifically, the designs of Cao et al.’s ACFS and Patterson et al.’s system are based on the simplifying assumption that block fetch time is fixed, independent of both the block’s location relative to the disk head and the block’s absolute location on disk. Based on this disk model, their derived optimal prefetching rules state that 1) blocks must be prefetched from disk in precisely application access order, and 2) a block must be prefetched as soon as there is available RAM. Because RAM was so scarce in systems of the time, this had the effect of retrieving data from disk just before it was needed, although Cao et al. also note that in practice request reordering can provide a significant performance improvement for the disk. Their implementation consists of one piece that takes great care to prefetch the very next block as soon as there is memory, and a second piece that buffers 4 to 16 of these requests to capitalize on disk ordering benefits. This approach is no longer the best trade-off; increased RAM sizes and larger performance gaps between disks and processors make it more important to maximize disk throughput. Libprefetch thus actively waits to request disk data until it can prefetch enough blocks to fill a significant portion of memory. For seek-limited applications on today’s systems, minimizing seek distances by reordering large numbers of blocks reduces I/O time and application runtime significantly more than prior approaches.

Both Patterson’s system and Cao’s ACFS explicitly addressed process coordination, especially important since their implementations replaced the operating system’s cache eviction policy. Libprefetch, in contrast, implements prefetching in terms of existing operating system mechanisms, so a less coordinated approach suffices. Existing operating system algorithms balance I/O among multiple processes, while libprefetch’s internal contention controller automatically detects and adapts to changes in available memory. In this regard, our approaches are complementary; something like ACFS’s two-level caching might further improve prefetching performance relative to Linux’s default policy.

Using a modest amount of RAM to cache prefetches, Patterson et al. and Cao et al. achieved maximum single-disk improvements of 55% (2.2x) and 49% (2x), respectively. Patterson et al. used multiple disks to achieve additional speedups, whereas libprefetch uses additional

RAM to achieve speedups of as much as 20x. For concurrent process performance Patterson et al. report a maximum performance improvement of 65% (2.9x), Cao et al. 76% (4.2x); we see improvements of 4x to 23x.

Inferred Accesses Rather than requiring the application to explicitly supply a list of future reads, a prefetching system can automatically generate the list—either from application source code, using static analysis [1, 4, 25, 26], or from the running application, using speculative execution [3, 7]. Static analysis can generate file read lists, but data dependence and analytic imprecision may limit these methods to simple constructs that do not involve abstractions over I/O. Speculative-execution prefetchers use spare CPU time to tell the operating system what file data will be needed. Speculation can provide benefits for unmodified applications, and is especially useful when it is difficult to programmatically produce the access pattern. Libprefetch could serve as the back end for a system that determined access patterns using analysis or speculation, but the less-precise information these methods obtain might reduce prefetching’s effectiveness relative to our results.

Libprefetch’s performance benefits are competitive with these other systems. For example, Chang et al. [3] focus on parallel disk I/O systems that provide more I/O bandwidth than is used by an unmodified application. Libprefetch obtains more prefetching benefit with one disk than Chang et al. find going to four.

Prefetching in Databases Lacking good OS support [21], applications like databases have long resorted to raw disk partitions to, for example, implement their own prefetch systems [22]. Better OS mechanisms, such as libprefetch, may reduce the need for this duplication of effort.

POSIX Asynchronous I/O The POSIX Asynchronous I/O [10], **posix_fadvise**, and **posix_madvise** [9] interfaces allow applications to request prefetching, but current implementations of these interfaces tend to treat prefetching advice as mandatory and immediate. To achieve good performance, applications must decide when to request a prefetch, how much to prefetch, and how to order requests. Libprefetch uses **posix_fadvise** as part of its implementation and manages these details internally.

3 The Impact of Modern Disk Characteristics on Prefetching

Disk prefetching algorithms aim to improve the performance of future disk reads by reading data before it is needed. Since our prefetching algorithm will use precise application information about future accesses, we need not worry about detecting access patterns or trying to

predict what the application will use next. Instead, the chief goal is to determine the fastest method to retrieve the requested data from disk.

This section uses disk benchmarks to systematically build up a prefetching algorithm that takes advantage of the strengths, and as much as possible avoids the weaknesses, of modern I/O subsystems. The prefetching algorithm must read at least the blocks needed by the application, so there are only a few degrees of freedom available. The prefetcher can reorder disk requests within the window of memory available for buffering, it can combine disk requests, and it can read non-required data if that would help. Different disk layout or block allocation algorithms could also lead to better performance, but these file system design techniques are orthogonal to the issues we consider.

3.1 Seek Performance

Conventional disk scheduling algorithms do not know what additional requests will arrive in the future, leading to a relatively small buffer of requests that can be reordered. In contrast, a prefetching algorithm that does know future accesses can use a reorder buffer as large as available memory. A larger buffer can substantially reduce average seek distance. In this section, we measure the actual cost of various seek distances on modern disks, aiming to determine where seek distance matters and by how much.

We measured the average time to seek various distances, both forward and backward. Because the seek operation is below the disk interface abstraction, it is only possible to measure a seek in conjunction with a read or write operation. Therefore, the benchmarks start by reading the first block of the disk to establish the disk head location (or the last block, if seeking backward), then read several blocks from the disk, each separated by the seek distance being tested. With this test methodology, a seek distance of zero means that we read sequential disk locations with multiple requests, and a seek distance of -1 block re-reads the same block repeatedly. All the tests in this section use Direct I/O to skip the buffer cache, ensuring that buffer cache hits do not optimize away the effects we are trying to measure. See the evaluation section for a description of our experimental setup.

The results of running this benchmark on two disks is shown in Figure 1. The “Read” lines in the graphs, included for comparison, show the time to read the given amount of data, assuming maximum throughput. In either direction, the cost of large ($\geq 1\text{MB}$) seeks is substantial, ranging from 5ms to 10ms; avoiding just 200 of these seeks could reduce runtime by one to two seconds. The cost is similar in either direction, except that seeks with distance less than 1MB are faster when seeking forward than seeking backward. This motivates the choice

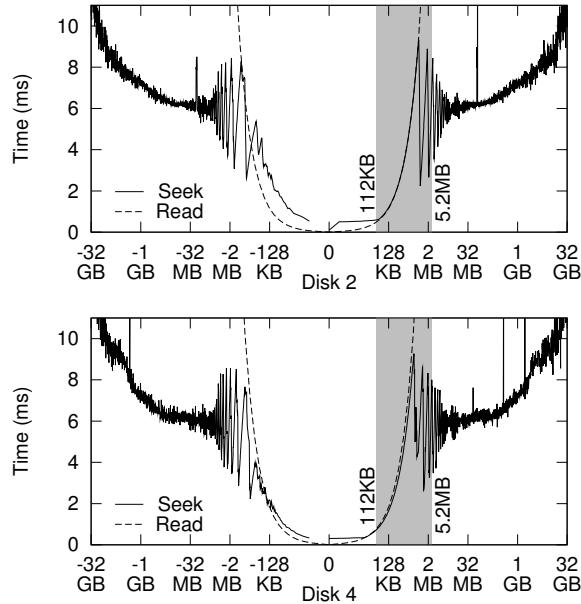


Figure 1: Average time to seek a given distance (forward or backward) compared to maximum read throughput. The oscillations are due to disk geometry combined with rotational latency. The grey region highlights where seek time changes most dramatically.

of a Circular LOOK algorithm: scan forward servicing requests until there are no requests past the head position, then return to the request with the lowest offset and repeat.

Seek time increases by roughly a factor of five from around 112KB to roughly 1 to 5MB, shown as the highlighted regions in the graphs. In contrast, seek times for distances above 5MB increase slowly (note the graph's log-scale x axis), by about a factor of two. Not considering the disk geometry effects visible as oscillations, a disk scheduling algorithm should minimize seek distance; though not all seek reductions are equal, reducing medium seeks far below 1MB will have more impact than reducing very large seeks to 1MB or more.

Figure 1 also shows the unexpected result that for distances up to 32KB, it may be cheaper to read that amount of data than to seek. This suggests that adjacent requests with small gaps might be serviced faster by requesting the entire range of data and discarding the uninteresting data. Our prefetching algorithm implements this feature, which we call *infill*.

3.2 Effect of Reorder Buffer Size

To reorder disk requests, the prefetch system must buffer data returned ahead of the application's needs. Thus, the window in which the prefetch system can reorder requests is proportional to the amount of memory it can use to store their results. So how much memory should be used to reorder prefetch data? Is there a threshold where more memory doesn't improve runtime substantially?

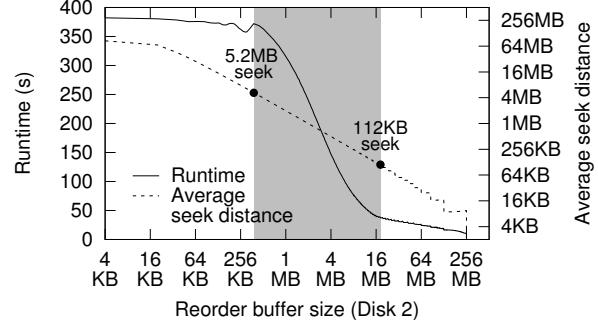


Figure 2: Effect of reorder buffer size on runtime and average seek distance for random reads. The test reads a total of 256MB of data. Along the x axis, we vary the reorder buffer size (the amount of data prefetched at once). Within each buffer, disk requests are sorted by logical block number. Runtime changes most dramatically in the grey region, where average seek distance drops from 5.2MB to 112KB, the boundaries of the highlighted region in Figure 1.

tially? The next benchmark tries to answer these questions by using various reorder buffer sizes. The benchmark is an artificial workload of 256MB of randomly chosen accesses to a 256MB file; some blocks in the file may be accessed multiple times and others may not be accessed at all, but the test uses Direct I/O so all the requests are satisfied from disk. Benchmarks for different amounts of total data had similar results.

The benchmark proceeds through the 256MB workload one reorder buffer at a time, reading pages within each buffer in C-LOOK order (that is, by increasing disk position). Figure 2 shows how the size of the reorder buffer affects runtime. The region of the graph between reorder buffer sizes of 384KB and 18MB, highlighted in grey, shows the most dramatic change in runtime. Examining the average seek distance of the resulting accesses helps to explain the change within the grey region. As the reorder buffer grows, the average seek distance in the grey region decreases from 5.2MB to 112KB (dotted line, right-hand y axis). This range of seek distances is greyed in Figure 1 and corresponds to the region where seek cost changes most dramatically.

However, increasing the reorder buffer beyond 18MB still has a substantial effect, decreasing runtime from 37.6 seconds to 8.4 seconds, an additional speedup of 4.7x. This continued decrease in runtime is due in part to the reduction in the number of disk passes needed to retrieve the data, from about 14 with a reorder buffer of 18MB down to one with a reorder buffer of 256MB. This demonstrates that the prefetching algorithm should prefetch as much as possible, but at least enough to reduce the average seek distance to 112KB, if possible.

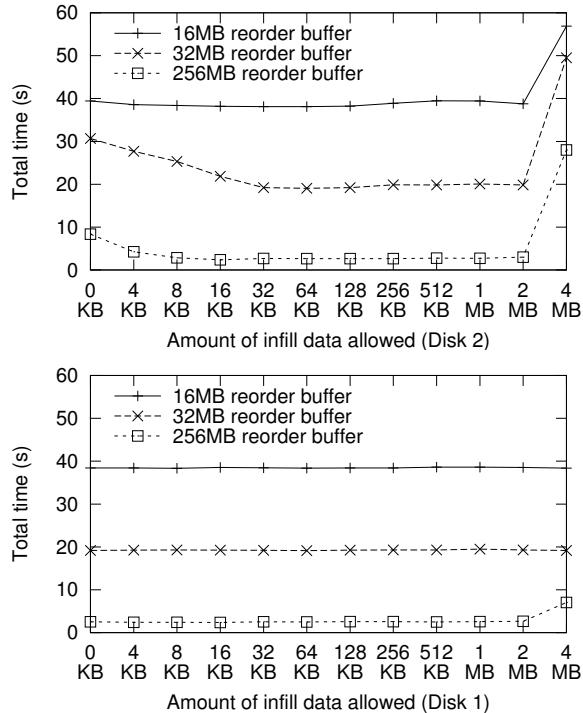


Figure 3: Effect of infill on runtime. The test reads a total of 256MB of data. Along the x axis, we vary the maximum amount of extra data read between requests. Infill of 32KB maximizes performance improvement; infill beyond 2MB can hurt performance.

3.3 Effect of Infill

Figure 1 suggested that reading and discarding small gaps between requests might be faster than seeking over those gaps. We modified the previous benchmark to add infill, varying the maximum infill allowed. Figure 3 shows that infill of up to 32KB reduces runtime on Disk 2. This corresponds to the region of Figure 1 where seeks take longer than similarly-sized maximum-throughput reads. Infill amounts between 32KB and 2MB have no additional effect, corresponding to the region of the seek graph where seek time is equal to read time for an equivalent amount of data. When infill is allowed to exceed 2MB, runtime increases, confirming that reads of this size are more expensive than seeking. For the 16MB reorder buffer dataset, infill has very little effect: as Figure 2 shows, the average seek distance for this test is 128KB, above the threshold where we expect infill to help. On Disk 1, however, infill was performance-neutral. Apparently its firmware makes infill largely redundant.

In summary, our tests show that infill amounts up to 32KB can help performance on some disks, but only for datasets that have a significant number of seeks less than 32KB in size.

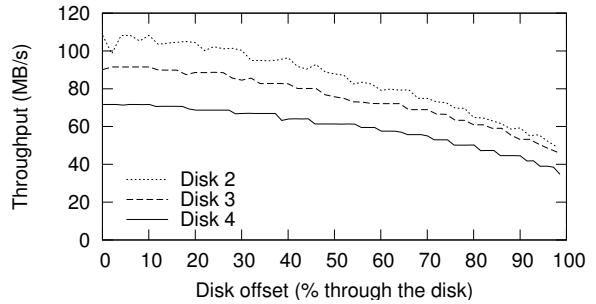


Figure 4: Read throughput at various disk offsets. This test reads 256MB at each sampled offset.

3.4 Other Techniques

Conventional wisdom says that read throughput is better at the outer edge of disk platters (low logical block numbers), which we confirm on several disks in Figure 4. The data for Disk 4 clearly shows the transitions between different areal densities (where the line steps down). The global effect is significant, slowing by about 50% from the first to the last logical block number. However, with large disks, even large files will usually span a small fraction of the disk, leading to a small speed difference between the beginning and end of a file.

It is also conventional wisdom that larger disk requests achieve better throughput. We tested this by reading a large amount of data (128MB) with a number of differently-sized read requests. Initially, this benchmark indicated that the size of read requests did not have an impact on performance, although performance differed when Linux's readahead was enabled. Upon examining block traces, we discovered that the request size issued to the disk remained constant, even as the application issued larger reads. With readahead enabled, large disk requests were always issued; with readahead disabled, disk requests were always 4KB in size. (It is odd that disabling readahead would cause Linux to always issue 4KB disk requests, even when the application requests much more data in a single read call; we call this a bug.) Using Direct I/O caused Linux to adhere to the request size we issued. Figure 5 shows that request sizes below 64KB do not achieve maximum throughput, although Linux's readahead code normally compensates for this.

3.5 Prefetching Algorithm

In summary, large reorder buffers lead to significant speed improvements, so prefetching should use as much memory as is available. Forward seeks can be faster than backward seeks, so prefetching should use a C-LOOK style algorithm for disk scheduling. Small amounts of infill can be faster than seeking between requests on some disks and have no negative effect on others, so infill should be used. The request size can have a significant impact on performance, but the I/O subsystem usually

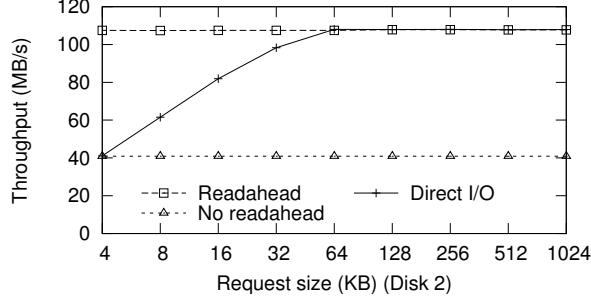


Figure 5: Effect of request size on throughput. Using or disabling readahead shows the two extremes of the performance spectrum. Direct I/O shows the true effect of request size on throughput.

optimizes this parameter sufficiently. Thus, libprefetch’s basic prefetching algorithm simply traverses an access list one reorder buffer at a time, prefetching each buffer’s data in disk order. Although further optimizations might be possible—for example, for some access patterns it may be more advantageous to prefetch a partial reorder buffer—we currently implement the basic version.

The tests also help us understand how prefetching can scale. The reorder buffer improves performance by reducing average seek distance, but not all reductions are equally valuable. For example, Figure 1 indicates that reducing average seek distance 500x from 1GB to 2MB wouldn’t improve performance much. The amount a given reorder buffer reduces average seek distance depends on access pattern and file system layout, but for uniform random accesses and sequential layout, we can approximate the resulting seek distance analytically. Given an N -page sequential file and a reorder buffer of K pages ($2 \leq K \ll N$), the expected average seek distance will be $2N(K-1)/K(K+1)$ pages.² This distance is roughly proportional to the ratio of file size to reorder buffer size. Thus, increasing memory by some factor will reduce average seek distance by the same factor, or, equivalently, produce the same seek distance when processing a proportionally bigger file. The formula can also be used to predict when seek distance will rise into the unproductive region of Figure 2. For instance, assuming 3GB of RAM available for prefetching and uniform random accesses, a 384GB file will achieve an average seek distance of 1MB, the rough “large seek” boundary.

4 Libprefetch

The libprefetch library implements our prefetching algorithm underneath a callback-based interface that easily integrates into applications. Libprefetch calls an application’s callback when it needs more prefetching information. The callback can compute the application’s list of future accesses and pass it to libprefetch’s **request_prefetching** function. The computed list can replace or augment the current access list.

```

struct access_entry {
    #define PF_APPEND      1
    #define PF_SET         2
    #define PF_SET_FROM_MARK 4
    #define PF_DONE        8
};

typedef void (*callback_t)(void * arg,
                         int lastMarkedFD, loff_t lastMarkedOffset,
                         int requestedFD, loff_t requestedOffset);
ssize_t request_prefetching(client_t c, const struct access_entry * a,
                           size_t n, int type);

client_t register_client(callback_t cb, void * arg);
int unregister_client(client_t c);

region_t register_region(client_t c, int fd, loff_t start, loff_t end);
int unregister_region(client_t c, region_t r);

int ignore_accesses(client_t c);
int unignore_accesses(client_t c);

```

Figure 6: Libprefetch interface. Applications create a client with **register_client**, then declare regions of files where accesses will be prefetched using **register_region**. When libprefetch needs an updated access list it calls back into the application with the registered callback function. This callback updates the access list with calls to **request_prefetching**. Finally, **ignore_accesses** lets the application read data from a prefetchable region without affecting the access list.

Libprefetch periodically asks the kernel to prefetch a portion of the access list; how much to prefetch depends on available memory. As the access list is consumed, or if actual accesses diverge from it, libprefetch calls back into the application to extend the list. Libprefetch tracks the application’s progress through the list by overriding the C library’s implementations of **read**, **ready**, and **pread**.

Figure 6 summarizes the libprefetch interface. The rest of this section discusses libprefetch in more detail, including a design rationale and important aspects of its implementation.

4.1 Callbacks

Libprefetch’s callback-based design achieves the following goals:

- The interface should minimize interference with application logic.
- The application should not have to guess when to make new prefetch requests. Therefore, libprefetch should actively request new prefetch information from the application. Lower-level components like libprefetch or the system’s buffer cache manager best know when prior prefetching results have completed, indicating the need for additional prefetching, or when a read request blocks, indicating that the application’s access list was inaccurate.

This model lets applications isolate most access-list management logic in a self-contained callback function. Of course, an application that prefers to actively manage the access list can do so.

Libprefetch issues a callback from its implementation of **read**, **ready**, or **pread**. The callback is passed sev-

eral arguments indicating the application’s position in the access list. (This further isolates prefetching from application logic, since the application need not explicitly track this position.) These arguments include a user-specified `void *`, the file descriptor and offset of the access that triggered the callback, and the file descriptor and offset of the most recently accessed *marked page*. Marked pages are application-specified access-list landmarks that can be more useful to the callback than the current position. For example, consider a database accessing data via an index. The index’s pages are accessed once each in a predictable, often sequential, order, but the data pages may be accessed seemingly randomly (and multiple times each, if the data set doesn’t fit in memory). This makes libprefetch’s position in the index portion of the access list more useful for planning purposes than its position in the data page portion. The database thus marks index pages within its access list, allowing its callback to quickly determine the most recently read index page, and therefore how far reading has progressed. Note that the most recent marked page need not have actually been read, as long as subsequent pages were read. This can happen, for example, when an index page was already in an application-level cache. Libprefetch correctly handles such small divergences from a predicted access list.

In addition to providing a means to specify the callback function, the `register_client` interface would let multiple threads within an application specify their own access lists. Our current implementation of libprefetch does not support multiple clients per process, though we expect this feature would be easy to implement for clients with non-overlapping file regions.

4.2 The Access List

The application interface for specifying future accesses was designed to achieve three goals:

- The basic interface for requesting prefetching should be the simplest sensible interface that can represent arbitrary access patterns, such as a list of future accesses in access order.
- Prefetching should work both within and across files.
- The application should be able to define its access pattern incrementally. The data structures required to specify a large pattern would take up memory that could otherwise be used for data. More fundamentally, some applications only gradually discover their access patterns.

The application specifies its access list by filling in an array of `access_entry` structures with the file descriptor and offset for each intended access. An arbitrary subset of these structures can be marked. If the application will access block *A*, then block *B*, and then block

A again, it simply adds those three entries to the array. It passes this array to libprefetch’s `request_prefetching` function. Each call to `request_prefetching` can either replace the current list, append to the list, or replace the portion of the current list following the most-recently-accessed marked entry. Libprefetch adjusts its idea of the application’s current position based on the new list. The number of accepted entries is returned; once libprefetch’s access-list buffer fills up, this will be less than the number passed in. When the application has transferred its entire access list to libprefetch, or libprefetch has indicated that its buffer is full, the application signals that it is done updating the list.

Libprefetch assumes that the sequence of file reads within registered file regions is complete—all read requests to registered regions should correspond to access-list entries. Accesses to non-registered regions are ignored; applications can manage these regions with other mechanisms. If the application accesses a file offset within a registered region but not in the access list, libprefetch assumes the application has changed the access plan and issues a callback to update the access list. However, the application can tell libprefetch to ignore a series of accesses. This is useful to avoid callback recursion: sometimes a callback must itself read prefetchable data while calculating the upcoming access list.

4.3 Callback Example

Figure 7 presents pseudocode for a sample callback, demonstrating how libprefetch is used. This callback is similar to the one used in our GIMP benchmark (Section 5.3). GIMP divides images into square regions of pixels called *tiles*; during image transformations, it iterates through the tiles in both row- and column-major order, both of which could cause nonsequential access. The pseudocode prefetches an image’s tile data in the current access order (`img->accessOrder`).

The callback first uses its arguments to determine the application’s position in its access pattern (lines 4–5). Next, it traverses tile information structures to determine future accesses (line 6). For each future access, the callback records the file descriptor, offset, and whether the access is considered marked (lines 7–9). Access entries accumulate in an array and are passed to libprefetch in batches (line 10). If libprefetch’s access-list buffer fills up, the callback returns (lines 12, 15–16). The callback’s first call to `request_prefetching` clears the old access list and sets it to the new value (line 3); subsequent calls append to the access list under construction (line 13). Finally, the callback informs libprefetch about any remaining access entries and signals completion (line 18).

```

void callback(state, markFd, markOffset, reqFd, reqOffset) {
1: struct access_entry accesses[BATCH_SIZE];
2: int accepted, full, n = 0;
3: int mode = PF_SET;
4: tileInfo_t *tile = getTileInfo(reqFd, reqOffset);
5: imageInfo_t *img = tile->imageInfo;

6: for ( ; !lastTile(tile); tile = nextTile(tile, img->accessOrder) ) {
7:     accesses[n].page_offset = tile->swap.offset;
8:     accesses[n].fd = img->swap.file;
9:     accesses[n++].marked = 0;
10:    if (n == BATCH_SIZE) {
11:        accepted = request_prefetching(state->client,
12:                                         accesses, n, mode);
13:        full = (accepted < n);
14:        mode = PF_APPEND;
15:        n = 0;
16:        if (full)
17:            break;
18:    }
19: request_prefetching(state->client, accesses, n,
20:                      mode | PF_DONE);
}

```

Figure 7: Pseudocode for a libprefetch callback function.

4.4 Interface Discussion

We evaluated several alternatives before arriving at the libprefetch interface. Previous designs’ deficiencies may illuminate libprefetch’s virtues.

The initial version of libprefetch stored several flags for each access-list entry, including a flag that indicated the page would be used only once (it should be evicted immediately after use). In several cases, after some unproductive debugging, we found that we had incorrectly set the flag. By tracking progress through the access list, the next iteration of libprefetch made it possible to automatically detect pages that aren’t useful in the short-term future, so we removed this ability to accidentally induce poor performance.

An earlier attempt to enhance prefetching tried to expand the methodology used by readahead, namely inferring future accesses from current accesses. Readahead infers future accesses by assuming that several sequential accesses will be followed by further sequential accesses. Our extension of this concept, **fdepend**, allowed the application to make explicit the temporal relationships between different regions of files. With **fdepend**, an application might inform the kernel that after accessing data in range *A*, it would access data in range *B*, but no longer access data in range *C*. This mechanism introduced problems that we later solved in libprefetch. For example, because it wasn’t clear which relationships would be useful in advance of a particular instant of execution, applications would specify all relationships up front. This caused large startup delays as an application enumerated all the relationships; in addition, storing all the relationships required substantial memory. Libprefetch’s access list is both simpler for applications to generate and easier to specify incrementally.

4.5 Implementation

Libprefetch is mostly implemented as a user-level library. This choice both demonstrates the benefits possible with minimal kernel changes and avoids end-runs around our operating system’s existing caching and prefetching policies. An application using libprefetch might issue different file-system-related system calls than the unmodified application, but as far as the operating system is concerned, it is doing nothing out of the ordinary. The kernel need not change its policy for managing different applications’ conflicting needs. (Nevertheless, a kernel implementation could integrate further with existing code, would have better access to buffer cache state and file system layout, and might offer speed advantages by reducing system call overhead.) Infill, however, is implemented as a kernel modification. Infill is not strictly a prefetching optimization, but a faster way to read specific patterns of blocks.

Each time libprefetch intercepts a **read** from the application, it first checks whether the read corresponds to a registered region. If so, it uses a new system call, **fincore**, to see whether the requested page(s) are already in the buffer cache. If there is a miss in the buffer cache, a page has been prematurely evicted or the application has strayed from its access list. In either case, libprefetch issues a new round of prefetch requests, possibly calling into the application first to update the access list. The **fincore** system call was inspired by **mincore**; it takes a file descriptor, an offset and length, and the address of a bit vector as input, and fills in the bit vector with the state of the requested pages of the file (in memory or not).

When libprefetch decides that it should prefetch more data, it first consults the contention controller (described below) to determine the size of its reorder buffer for this round of prefetching. Then it walks the access list until it has seen the appropriate number of unique pages. Once the set of pages to be prefetched is determined, libprefetch evicts any pages from the previous round of prefetching that are not in the current prefetch set and asks the kernel to prefetch, in file offset order, the new set of pages. This process is where libprefetch differs most from previous prefetching systems. Instead of overlapping I/O and CPU time, libprefetch blocks the application while it fetches many disk blocks. This makes sense, particularly for nonsequential access patterns, because sorted and batched requests are usually faster than in-order requests. Prefetching from a separate thread would allow the main thread to continue once its next request was in memory, but as soon as the main thread made a request located near the end of the sorted reorder buffer, it would block for the rest of the prefetching phase anyway.

Libprefetch makes prefetch requests using **posix-fadvise**, a rarely-used system call that does for files

what **madvice** does for memory. It takes a file descriptor, an offset and length, and “advice” about that region of the file. Libprefetch uses two pieces of advice: `POSIX_FADV_WILLNEED` informs the kernel that the given range of the file should be brought into the buffer cache, and `POSIX_FADV_DONTNEED` informs it that the given range of the file is no longer needed and can be dropped from the buffer cache.

We found some weaknesses in the Linux implementation of **posix_fadvise**. The `WILLNEED` advice has no effect on file data that is already in memory; for example, if a given page is next on the eviction list before `WILLNEED` advice, it will still be next on the eviction list after that advice. We therefore changed **posix_fadvise** to move already-in-memory pages to the same place in the LRU list they would have been inserted had they just come from disk. The implementation of `DONTNEED` also has pitfalls. Linux suggests applications flush changes before issuing `DONTNEED` advice because dirty pages are not guaranteed to be evicted. However, it tries to assist with this requirement by starting an asynchronous write-back of dirty data in the file upon receiving `DONTNEED` advice. Unfortunately, it starts this write-back even if the to-be-evicted data is not dirty. Because of this unexpected behavior, we found it faster to `DONTNEED` pages in large batches instead of incrementally.

Our modifications supply **posix_fadvise** with additional functionality: libprefetch uses the system call to intentionally reorder the buffer cache’s eviction list. Libprefetch already uses `DONTNEED` advice to discard pages it no longer needs, but ordering the eviction list further improves performance in the face of memory pressure. After prefetching a set of pages in disk order, libprefetch again advises the kernel that it `WILLNEED` that data, but in reverse access order. If memory pressure causes some pages to be evicted, the evicted pages will now be those needed furthest in the future. This enables the application to make as much progress as possible before libprefetch must re-prefetch evicted pages.³

Together, **fincore** and our modified **posix_fadvise** give libprefetch enough access to buffer cache state to work effectively. **fincore** lets libprefetch query the state of particular pages, and **posix_fadvise** lets it bring in pages from disk, evict them from the buffer cache, and reorder the LRU list. Since these operations are all done from user space, existing kernel mechanisms can account for resource usage and provide fairness among processes. An efficient system call that translates file offsets to the corresponding on-disk block numbers would improve libprefetch’s support for nonsequential file layouts.

Libprefetch is approximately 2,400 lines of commented code, including several disabled features that showed no benefit. The kernel changes for **posix_fadvise** and **fincore** are 130 lines long.

4.6 Concurrent Execution

As described so far, libprefetch monopolizes both disk bandwidth and the buffer cache. Of course, this behavior could seriously degrade other applications’ performance. In this section we discuss modifications to libprefetch that improve fairness and application performance even with multiple uncoordinated applications running concurrently.

Disk contention is the easier problem to solve: Linux’s default fairness mechanisms work effectively as is. From the point of view of the operating system, disk contention caused by libprefetch is indistinguishable from contention caused by any other application. Further disk access coordination could yield better performance, but might raise other issues, such as fairness, denial of service, and security concerns.

The buffer cache, however, requires a different approach. As discussed, libprefetch should use the maximum buffer cache space available. An early implementation simply queried the operating system for the amount of buffer cache space and used half of it (half showed the best performance in experiments). However, this did not account for other applications using the buffer cache, and in benchmarks with more than one application libprefetch’s attempt to dominate the buffer cache severely degraded performance. The current libprefetch explicitly addresses buffer-cache contention, but manages to do so without explicit coordination among processes. The key insight is that buffer cache management can be reformulated as a congestion-control problem. Libprefetch uses an additive-increase, multiplicative-decrease (AIMD) strategy, also used by TCP for network congestion control, to adapt to changes in available memory.

Libprefetch infers a contention signal when it finds that some of the pages it prefetched have been prematurely evicted. This should happen only under memory pressure, so libprefetch lowers its reorder buffer size with a multiplicative decrease. Conversely, when libprefetch consumes all of the pages it prefetched without any of them being prematurely evicted, it increases its reorder buffer size by an additive constant.

Because libprefetch knows the maximum size of the buffer cache (from the `/proc` file system), it starts out using most of available cache space, instead of using a slow-start phase. We always limit libprefetch’s reorder buffer to 90% (experimentally determined) of the RAM available for the buffer cache. Furthermore, since the contention controller can only adjust the reorder buffer size during a round of prefetching, it is somewhat aggressive in its upward adjustment, adding 10% of the maximum buffer cache space to the reorder buffer size; it halves the reorder buffer size to decrease. A quick evaluation of alternative values shows that libprefetch is not particularly sensitive to AIMD constants. The result

performs well: concurrently-running libprefetch-enabled applications transparently coordinate to achieve good performance, and libprefetch applications do not significantly degrade the performance of other applications.

4.7 Disk Request Infill

To implement infill, we modified the general Linux I/O scheduler framework and its pluggable CFQ scheduler (about 800 lines of changes). Linux's I/O schedulers already have the ability to merge adjacent requests, but they cannot merge non-adjacent ones. Upon receiving a new request, Linux looks for a queued request to merge with the incoming one. If no request is found, our modified scheduler then looks for the queued request nearest the new one. If the nearest request is within the maximum infill distance, we create a dummy request and merge it with the queued request. The incoming request is now adjacent to the expanded request and the two are merged.

While the I/O scheduler has infill requests in its queue, a new request may arrive that overlaps an infill request. The scheduling framework did not handle overlapped requests, so we explicitly link new requests to any queued infill requests that they overlap. When the infill request completes, the new request is serviced from the infill request's data.

5 Evaluation

In this section we evaluate libprefetch's impact on three different kinds of workloads: sequential, “strided,” and nonsequential. Linux's readahead mechanism already does a good job of optimizing sequential workloads; for these we expect at most a modest improvement with libprefetch. A *strided* workload consists of groups of sequential accesses separated by large seeks. Linux does not specifically try to detect strided access patterns and does little to improve their performance (unless the run of sequential accesses is substantial). Readahead is of no help to nonsequential access patterns, and we expect the biggest improvement with this workload. Our strided and nonsequential access patterns come from benchmarks of real applications, namely the GNU Image Manipulation Program (GIMP) and the SQLite database. This shows that libprefetch significantly improves real application performance.

When evaluating these workloads, we vary the amount of data accessed relative to the amount of RAM, showing how relative reorder buffer size affects prefetching improvements. In addition, we examine the impact of infill and the performance of multiple uncoordinated applications running concurrently.

5.1 Methodology

All the benchmarks in this section were run on a Dell Precision 380 with a 3.2GHz Pentium 4 CPU with 2MB

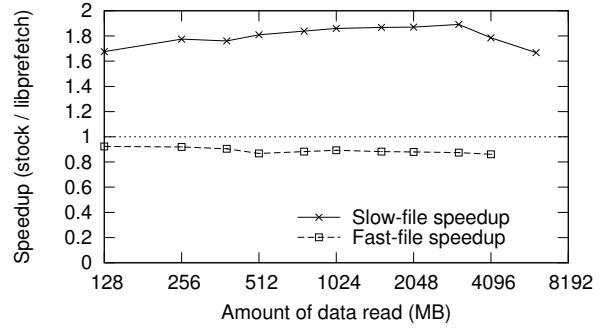


Figure 8: Speedup with libprefetch when reading a file sequentially. Due to file layout issues, some files can be retrieved more quickly; this benchmark shows both a fast and slow file. The slowdown on the fast file, 6 seconds for the 4GB test, is mostly due to libprefetch's CPU overhead. The slow file runtimes range from 6 s to 365 s for stock and 6 s to 220 s for libprefetch.

of L2 cache (hyperthreading disabled), a Silicon Image 3132-2 SATA Controller, and 512MB of RAM. Log output was written to another machine via sshfs. Tests use a modified Linux 2.6.20 kernel on the Ubuntu v8.04 distribution. The small size of main memory was chosen so that our tests of stock software would complete in a reasonable amount of time (a single stock 4GB SQLite test takes almost ten hours). As mentioned in Section 3.5, we believe libprefetch's speedup relative to unoptimized accesses is constant for a given ratio of data set size to prefetching memory, at least for uniform random accesses. Unless otherwise noted, these tests used Disk 1, a 500GB 7200 RPM SATA2 disk with a 32MB buffer and 8.5ms average seek time (Seagate ST3500320AS, firmware SD1A). Disk 2 is Disk 1 with older firmware, version SD15.

Some of the benchmarks in Section 3 used additional disks. Disk 3 is a 500GB 7200 RPM SATA2 disk with a 16MB buffer and 8.9ms average seek time (Western Digital WD5000AAKS) in a HP Pavilion Elite D5000T with a 2.66GHz Core 2 Quad Q9450 and 8GB of RAM, and Disk 4 is a 320GB 7200 RPM SATA2 disk with a 16MB buffer and 8.5ms average seek time (Seagate ST3320620AS) in a Dell Optiplex GX280 with a 2.8GHz Pentium 4 CPU, 1MB of L2 cache (hyperthreading disabled), a Silicon Image 3132-2 SATA Controller, and 512MB of RAM.

5.2 Sequential Access

Our sequential benchmark program reads a file from beginning to end. It is similar to `cat file > /dev/null`, but is libprefetch-enabled and can change various Linux readahead options. We observed, and confirmed with `dd`, great variance in sequential read performance on different files, from 20MB/s to 110MB/s. These differences are due to file fragmentation. For example, the fastest file has a significantly longer average consecutive block run

than the slowest file (3.8MB vs. 14KB).

For the slowest file, Figure 8 shows that libprefetch achieves improvements similar to previous prefetching work. We believe that Linux readahead is slower than libprefetch in this case because libprefetch sends many more requests to the disk scheduler at once, giving it more opportunity to reorder and batch disk requests. With the fast file, libprefetch is slightly slower than readahead. Examining the system and user time for the tests shows that the majority of the difference can be attributed to the additional CPU overhead that libprefetch incurs, which we have not yet tried to optimize.

5.3 Strided Access

Our strided benchmarks use the GNU Image Manipulation Program (GIMP) to blur large images, a workload similar to common tasks in high-resolution print or film work. GIMP divides the image into square tiles and processes them in passes, either by row or by column. When GIMP’s memory requirement for tiles grows beyond its internal cache size, it overflows them to a swap file. The swap file access patterns manifest themselves as strided disk accesses when a row pass reads the output of a column pass or vice versa. To blur an image, GIMP makes three strided passes over the swap file. While it is feasible to correctly detect and readahead these kinds of access patterns, Linux does not attempt to do so.

We changed the GIMP tile cache to allow processing functions to declare their access patterns: they specify the order (row or column) in which they will process a set of images. Multiple image passes are also expressible. We exposed this interface to GIMP plugins and modified the blur plugin to use this infrastructure (the modification was simple). The core GIMP functions use a common abstraction to make image passes, which we also modified to use the access pattern infrastructure. We changed a total of 679 lines: 285 for the plugin architecture, 40 to specify patterns in the blur plugin, 11 to alter the core image pass abstraction, and 343 to implement the libprefetch callback.

We benchmarked the time to blur a square RGBA image of the given size. Blur uses two copies of the image for most of the operation and three to finish, so memory requirements are higher than just the raw image size. We set the GIMP’s internal cache to 100MB to prevent significant amounts of double buffering in the GIMP and the operating system’s buffer cache; we do not use less than 100MB so that GIMP’s internal cache can contain up to three working image rows or columns for our largest test image. GIMP mallocs space for its file cache, so libprefetch does not attempt to use any of that 100MB of memory per GIMP instance. The GIMP benchmark is read/write, whereas our other benchmarks are read-only.

The results are shown in Figure 9. When the image

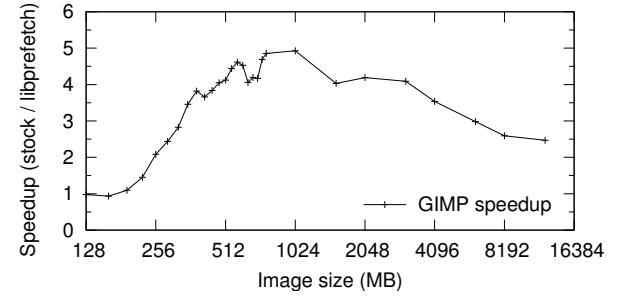


Figure 9: Speedup with libprefetch for GIMP to blur various sized images. When GIMP uses the disk, libprefetch reduces runtime by a factor of 2 to 5. Stock runtimes range from 28 seconds for the 128MB workload to almost 7.5 hours for the 12GB workload; the intermediate size of 1GB takes 38 minutes. The libprefetch runtime also starts at 28 seconds, but only climbs to 3 hours; the 1GB size takes 5 minutes.

size is small, all the data is held in the GIMP’s internal cache and the operating system’s buffer cache, so there is no disk access to optimize; stock and libprefetch runtimes are equal. As the image size increases from 192MB to 1GB, disk access increases and libprefetch achieves greater speedups. Libprefetch retrieves data from multiple rows (columns) before striding to the next part of those rows (columns). This amortizes the cost of the strided access pattern across the retrieval of multiple rows (or columns), achieving a speedup of up to 5x. As the image size increases, however, the number of rows or columns that can be retrieved in one pass decreases. The results for images greater than 1GB in size show this gradual decrease in speedup.

5.4 Nonsequential Access

Our nonsequential benchmark issues a query to a SQLite database. The dataset in the database is TPC-C like [6] with the addition of a secondary index by zip code on the **customer** table. We used datasets with 7 to 218 warehouses, yielding sizes between 132MB and 4110MB for the combination of the customer table and zip code index. Additionally, we configured SQLite to use 4KB pages (instead of the default 1KB pages) to match the storage unit and reduce false sharing.

The benchmark performs the query **SELECT * FROM customer ORDER BY c.zip**. (Runtime on this query was within a few percent on stock SQLite and stock MySQL.) For this query, each resulting row will be in a random file location relative to the previous row, inducing a large number of seeks. Consequently, we expect the query to have poor performance. When the dataset fits entirely in memory, each disk page only needs to be read once, after which the rest of the workload will be serviced from the buffer cache. However, if the dataset is larger than memory, pages will be read from disk multiple times (each page holds multiple rows).

The SQLite callback for libprefetch examines the off-

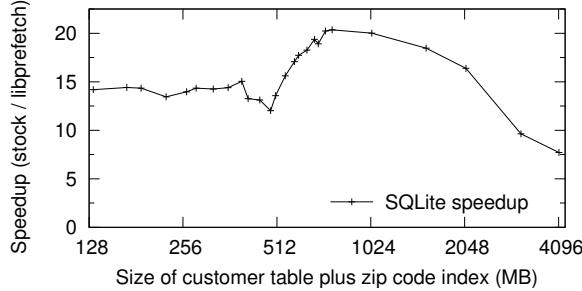


Figure 10: Speedup with libprefetch for SQLite when scanning a table by a secondary index. The initial 14x approximate speedup peaks at 20.3x, then falls to 7.7x. Stock runtime starts at about 3 minutes, climbs to 100 minutes by the 1GB test, and runs for nearly 10 hours for the 4GB test. Libprefetch’s runtime starts at 12 seconds and is only 77 minutes for the 4GB test; the 1GB test takes 7.5 minutes.

set for the current position and determines if it belongs to an index. If so, the callback iterates the index and tells libprefetch about the table data that the index points to. From then on, the callback marks the index pages and uses them to track its progress. This modification adds less than 500 lines of code.

Figure 10 shows the speedups for the nonsequential SQLite benchmark with 132MB to 4GB of data. The initial improvement of roughly 14x is because libprefetch is able to load the entire dataset in sequential order, whereas SQLite loads the data on demand (in random order) as it traverses the zip code index. As the dataset approaches the size of memory, the speedup decreases because libprefetch starts to require multiple passes over the disk. Then, between roughly 512MB and 768MB, we see a sharp increase in speedup: stock SQLite requires progressively more time due to a sharp decline in the buffer cache hit ratio as the dataset size exceeds the size of memory. As the dataset grows even larger, the density of libprefetch’s passes over the disk decreases, causing higher average seek distance and decreasing benefits from libprefetch.

Libprefetch processed the 4GB data set in just under 77 minutes. This is roughly 400 times slower than the 128MB data set, which took 11.6 s; processing time increased by about 13x more than data set size. Some expensive seeks are simply unavoidable. However, the libprefetch time is still 7.7x faster than stock SQLite.

5.5 Infill

Infill has no significant effect on the sequential or strided benchmarks because they have few infill opportunities. The large seeks in the strided access pattern are too large for infill to be a win. Similarly, stock SQLite with infill shows no significant speedup; the gaps between most requests are too large for infill to apply.

The effect of infill on SQLite when using libprefetch is shown in Figure 11. Disk 2 shows a substantial im-

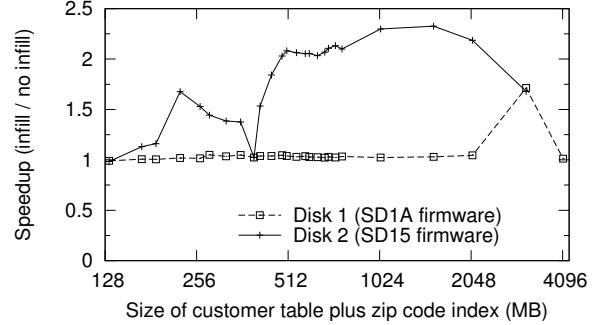


Figure 11: Speedup due to infill for Disk 1 and Disk 2. All tests used libprefetch. A firmware upgrade mostly alleviated the need for infill, though a modest effect is still observed.

provement for many of the tests, up to 2.3x beyond the speedups that libprefetch achieves. Libprefetch’s reordering shrinks the average gap between requests to the point where infill can improve performance. But as the dataset size increases, the density of requests in a given libprefetch pass across the disk decreases; as a result, infill’s applicability also decreases.

As noted earlier, Disk 1 and Disk 2 are the same disk: Disk 1 has newer firmware. While there is a noticeable difference in the infill speedup on these two disks, the difference in runtime when both libprefetch and infill are used is less than 10%. It appears that the updated firmware takes advantage of the hardware effect that leads to infill being useful. The 3GB test is somehow an exception, achieving a 1.7x speedup from infill on Disk 1. Except for that point, the maximum speedup from infill for Disk 1 is 1.049x. That is on par with the maximum infill speedup we saw on Disk 3, 1.084x.

Infill will never be helpful for some access patterns because there simply isn’t any opportunity to apply it. For other access patterns, when infill is used with libprefetch, it can either dramatically increase performance or provide a modest improvement, depending on the disk. None of our experiments showed a substantial negative impact from infill.

5.6 Concurrent Applications

We evaluated libprefetch’s effect on concurrent workloads first by running multiple concurrent instances of our benchmarks. Each instance used its own data file, so prefetching in one instance didn’t help any other. Figure 12 shows the runtime for 1 to 3 concurrent executions of both the SQLite and GIMP benchmarks (512MB datasets). Both with and without libprefetch, the runtime of the GIMP benchmark scales with the number of instances. The libprefetch versions run 3x to 4x faster than the stock versions. Stock SQLite scales more slowly than the number of instances; two SQLite instances take over 7x as long as a single instance, and three take more than 13x as long as one. Libprefetch improves SQLite’s scal-

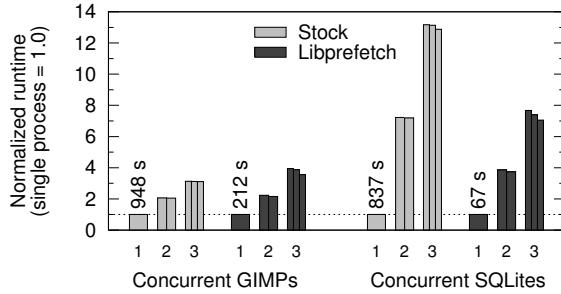


Figure 12: Stock vs. libprefetch application performance for one, two, and three application instances running concurrently. To highlight scaling behavior, times within each application group are normalized to the performance of a single application instance.

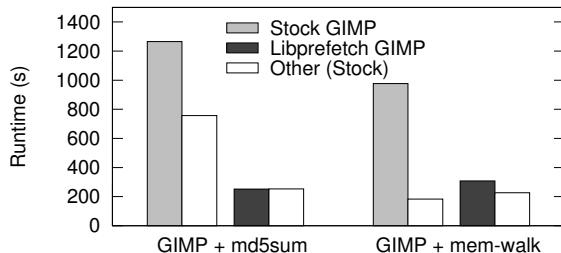


Figure 13: Runtime of GIMP, with and without libprefetch, measured concurrently with CPU- and memory-intensive microbenchmarks.

ing behavior; two SQLites take nearly 4x as long as one, and three take almost 8x as long as one. The libprefetch speedup over stock SQLite is 23.5x for two and 21.5x for three concurrent instances.

While libprefetch SQLite scales better than stock, libprefetch GIMP does not. We believe this is due to the amount of memory available for reorder buffers. Whereas each SQLite process occupies about 20MB of memory, each GIMP process occupies about 150MB (predominantly for its tile cache). On a machine with 500MB of memory available after bootup and three test processes, the memory available for reorder buffers is less than 50MB for GIMP versus 440MB for SQLite.

We also confirmed that libprefetch’s AIMD contention controller has the intended effect. With the contention controller disabled, each libprefetch process tried to use the entire buffer cache, causing many pages to be evicted before use. The tests ran for several times the stock runtime before we gave up and killed them. The AIMD mechanism is effective and necessary with our approach to contention management.

We tested resource contention more directly by running GIMP and SQLite concurrently with two resource-heavy benchmarks, md5sum and mem-walk. Md5sum calculates the MD5 checksum of a 2.13GB file; mem-walk allocates 100MB of memory and then reads each page in turn, cycling through the pages for a specified number of iterations. Figure 13 shows the runtime for these two microbenchmarks run concurrently with

GIMP, both with and without libprefetch. When running md5sum and GIMP concurrently, md5sum is faster with the libprefetch-enabled version of the GIMP. This is because the libprefetch-enabled GIMP has lower disk utilization, yielding more time for md5sum to use the disk.

An opposite effect comes into play when running the GIMP concurrently with the mem-walk benchmark. Since the same amount of CPU time is spent over a shorter total time, GIMP with libprefetch has a higher CPU utilization. Mem-walk takes about 25% longer with the libprefetch-enabled GIMP because it is scheduled less frequently. This slowdown is not specific to libprefetch; any CPU-intensive application would have a similar effect. The libprefetch speedup that GIMP gets with mem-walk is not as high as with md5sum, partly due to higher CPU contention and partly due to the smaller amount of memory available to libprefetch. Results for md5sum and mem-walk run concurrently with SQLite are similar.

6 Conclusion

An analysis of the performance characteristics of modern disks led us to a new approach to prefetching. Our prefetching algorithm minimizes the number of expensive seeks and leads to a substantial performance boost for nonsequential workloads. Libprefetch, a relatively simple library that implements this technique, can speed up real-world instances of nonsequential disk access, including image processing and database table scans, by as much as 4.9x and 20x, respectively, for workloads that do not fit in main memory. Furthermore, a simple contention controller enables this new prefetching algorithm to peacefully coexist with multiple instances of itself as well as other applications.

Acknowledgments

The authors thank the anonymous reviewers for their valuable feedback, and our shepherd, Geoff Kuenning, who was very generous with his time. This work was supported by the National Science Foundation under grants NSF-0430425, NSF-0427202, and NSF-0546892. Eddie Kohler is also supported by a Microsoft Research New Faculty Fellowship and a Sloan Research Fellowship.

Notes

¹For instance, the Apple Hard Disk 20SC, introduced in 1985, had an average access time of 65 to 85 msec and a maximum transfer speed of 1.25 MB/s [11]. The specifications for our disk (Seagate Barracuda 7200.11) quote an average access time of 4.16 msec and sustained transfer speed up to 105 MB/s [24].

²This formula was derived using uniform random real numbers, ignoring quantization effects, and is most precise when $5 \leq K \ll N$.

³Because prefetching and readahead are speculative, prefetched pages are inserted into the LRU list at a lower priority (at the head

of the inactive list) than pages that were explicitly **read**. This can lead to discrepancies between the LRU list order and the access list order.

References

- [1] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, May 2001.
- [2] Pei Cao, Edward Felten, Anna Karlin, and Kai Li. Implementation and performance of integrated application-controlled caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.
- [3] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 1–14, New Orleans, Louisiana, February 1999.
- [4] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Department of Computer Science Tech Report PCS-TR94-243, Dartmouth College, November 1994.
- [5] Microsoft Corp. *SuperFetch*, 2006. <http://www.microsoft.com/windows/windows-vista/features/superfetch.aspx>.
- [6] Transaction Processing Performance Council. *TPC-C Online Transaction Processing Benchmark*, April 2009. <http://www.tpc.org/tpcc/>.
- [7] Keir Fraser and Fay Chang. Operating system I/O speculation: How two invocations are faster than one. In *Proc. 2003 USENIX Annual Technical Conference*, pages 325–338, San Antonio, Texas, June 2003.
- [8] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proc. USENIX Summer 1994 Technical Conference*, pages 197–207, Boston, MA, June 1994.
- [9] IEEE. *POSIX 1003.1-2001*, 2001.
- [10] IEEE. *POSIX 1003.1b*, 1993.
- [11] Apple Inc. Apple Hard Disk 10SC: Specifications (Discontinued), November 2008. <http://docs.info.apple.com/article.html?artnum=1931>.
- [12] Apple Inc. HFS plus volume format. Section: Hot files. Technical Note TN1150, March 2004.
- [13] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian N. Bershad, Pei Cao, Edward W. Felten, Garth A. Gibson, Anna R. Karlin, and Kai Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 19–34, Seattle, Washington, October 1996.
- [14] Thomas M. Kroeger and Darrell D. E. Long. The case for efficient file access pattern modeling. In *Proc. 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 14–19, Rio Rico, AZ, March 1999.
- [15] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *Proc. 1997 USENIX Annual Technical Conference*, pages 275–288, Anaheim, California, January 1997.
- [16] Mark Palmer and Stanley B. Zdonik. FIDO: A cache that learns to fetch. In *Proc. 17th International Conference on Very Large Data Bases (VLDB '91)*, pages 255–264, Barcelona, Catalonia, Spain, September 1991.
- [17] R. Hugo Patterson and Garth A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proc. 3rd International Conference on Parallel and Distributed Information Systems (PDIS '94)*, pages 7–16, Austin, TX, 1994.
- [18] R. Hugo Patterson, Garth A. Gibson, Eka Gintin, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, CO, December 1995.
- [19] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.
- [20] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On multidimensional data and modern disks. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 225–238, San Francisco, CA, December 2005.
- [21] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.
- [22] Michael Stonebraker, John Woodfill, Jeff Ransome, Marguerite Murphy, Marc Meyer, and Eric Allman. Performance enhancements to a relational database system. *ACM Transactions on Database Systems*, 8(2):167–185, June 1983.
- [23] Carl Tait and Dan Duchamp. Detection and exploitation of file working sets. In *Proc. 11th International Conference on Distributed Computing Systems*, pages 2–9, Arlington, TX, May 1991.
- [24] Seagate Technology. ST3500320AS - Barracuda 7200.11 SATA 3Gb/s 500-GB Hard Drive, April 2009. <http://www.seagate.com/ww/v/index.jsp?vgnnextoid=c89ef141e7f43110VgnVCM100000f5ee0a0aRCRD>.
- [25] Rajeev Thakur, Rajesh Bordawekar, and Alok Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. *ACM SIGARCH Computer Architecture News*, 22(4):23–28, September 1994.
- [26] Kishor S. Trivedi. An analysis of prepaging. *Computing*, 22(3):191–210, September 1979.
- [27] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, September 1996.

Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances

Anton Burtsev[†], Kiran Srinivasan, Prashanth Radhakrishnan,
Lakshmi N. Bairavasundaram, Kaladhar Voruganti, Garth R. Goodson

[†]*University of Utah*

NetApp, Inc.

aburtsev@flux.utah.edu, {skiran, shanth, lakshmib, kaladhar, goodson}@netapp.com

Abstract

Enterprise-class server appliances such as network-attached storage systems or network routers can benefit greatly from virtualization technologies. However, current inter-VM communication techniques have significant performance overheads when employed between highly-collaborative appliance components, thereby limiting the use of virtualization in such systems. We present *Fido*, an inter-VM communication mechanism that leverages the inherent relaxed trust model between the software components in an appliance to achieve high performance. We have also developed common device abstractions - a network device (MMNet) and a block device (MMBlk) on top of Fido.

We evaluate MMNet and MMBlk using microbenchmarks and find that they outperform existing alternative mechanisms. As a case study, we have implemented a virtualized architecture for a network-attached storage system incorporating Fido, MMNet, and MMBlk. We use both microbenchmarks and TPC-C to evaluate our virtualized storage system architecture. In comparison to a monolithic architecture, the virtualized one exhibits nearly no performance penalty in our benchmarks, thus demonstrating the viability of virtualized enterprise server architectures that use Fido.

1 Introduction

Enterprise-class appliances [4, 21] are specialized devices providing services over the network to clients using standardized protocols. Typically, these appliances are built to deliver high-performance, scalable and highly-available access to the exported services. Examples of such appliances include storage systems (NetApp [21], IBM [14], EMC [8]), network-router systems (Cisco [4], Juniper [16]), etc. Placing the software components of such appliances in separate virtual machines (VMs) hosted on a hypervisor [1, 25] enables multiple benefits —fault isolation, performance isolation, effective resource utilization, load balancing via VM migration,

etc. However, when collaborating components are encapsulated in VMs, the performance overheads introduced by current inter-VM communication mechanisms [1, 17, 26, 28] is prohibitive.

We present a new inter-VM communication mechanism called *Fido* specifically tailored towards the needs of an enterprise-class appliance. Fido leverages the relaxed trust model among the software components in an appliance architecture to achieve better performance. Specifically, Fido facilitates communication using read-only access between the address spaces of the component VMs. Through this approach, Fido avoids page-mapping and copy overheads while reducing expensive hypervisor transitions in the critical path of communication. Fido also enables end-to-end zero-copy communication across multiple VMs utilizing our novel technique called *Pseudo Global Virtual Address Space*. Fido presents a generic interface, amenable to the layering of other higher-level abstractions. In order to facilitate greater applicability of Fido, especially between components developed by different collaborating organizations, Fido is non-intrusive, transparent to applications and dynamically pluggable.

On top of Fido, we design two device abstractions, *MMNet* and *MMBlk*, to enable higher layers to leverage Fido. MMNet (*Memory-Mapped Network*) is a network device abstraction that enables high performance IP-based communication. Similarly, MMBlk is a block device abstraction. MMNet performs consistently better on microbenchmarks in comparison to other alternative mechanisms (XenLoop [26], Netfront [1] etc) and is very close in performance to a loopback network device interface. Likewise, MMBlk outperforms the equivalent open-source Xen hypervisor abstraction across several microbenchmarks.

As a case study, we design and implement a full-fledged virtualized network-attached storage system architecture that incorporates MMNet and MMBlk. Microbenchmark experiments reveal that our virtualized

system does not suffer any degradation in throughput or latency in most test cases as compared to a monolithic storage server architecture. TPC-C macrobenchmark results reveal that the difference in performance between our architecture and the monolithic one is almost imperceptible.

To summarize, our contributions are:

- A high-performance inter-VM communication mechanism - Fido, geared towards software architectures of enterprise-class appliances.
- A technique to achieve end-to-end zero-copy communication across VMs - *Pseudo Global Virtual Address Space*.
- An efficient, scalable inter-VM infrastructure for connection management.
- Two high-performance device abstractions (MMNet and MMBlk) to facilitate higher level software to leverage the benefits of Fido.
- A demonstration of the viability of a modular virtualized storage system architecture utilizing Fido.

The rest of the paper is organized as follows. In Section 2, we present the background and the motivation for our work. Section 3 discusses the design and implementation of Fido and the abstractions - MMNet and MMBlk. Next, we evaluate Fido and the abstractions using standard storage benchmarks in Section 4. A case study of a network attached storage system utilizing Fido is presented in Section 5. In Section 6, we discuss related work. Finally, in Section 7 we present our conclusions.

2 Background and Motivation

In this section, we first provide an overview of appliance architectures and the benefits of incorporating virtualization in them. Next, we present the performance issues in such virtualized architectures, followed by a description of existing inter-VM communication mechanisms and their inadequacy in solving performance issues.

2.1 Enterprise-class Appliance Architectures

We are primarily concerned about the requirements and applicability of virtualization technologies to enterprise-class server appliances. Typically, these appliances provide a specialized service over the network using standardized protocols. High-performance access and high-availability of the exported network services are critical concerns.

Enterprise appliances have some unique features that differentiate them from other realms in which virtualization technologies have been adopted aggressively. In particular, the software components in such an architecture are extremely collaborative in nature with a large amount of data motion between them. This data flow is often organized in the form of a pipeline. An example

of an enterprise appliance is a network-attached storage system [21, 27] providing storage services over standardized protocols, such as NFS and CIFS. Such a storage system consists of components such as a protocol server, a local file system, software RAID, etc. that operate as a pipeline for data.

2.2 Virtualization Benefits for Appliances

Virtualization technologies have been highly successful in the commodity servers realm. The benefits that have made virtualization technologies popular in the commodity server markets are applicable to enterprise-class server appliances as well:

- **High availability:** Components in an enterprise appliance may experience faults that lead to expensive disruption of service. Virtualization provides fault isolation across components placed in separate VM containers, thereby enabling options such as micro-reboots [2] for fast restoration of service, leading to higher availability.
- **Performance isolation/Resource allocation:** Virtualization allows stricter partitioning of hardware resources for performance isolation between VMs. In addition, the ability to virtualize resources as well as to migrate entire VMs enables the opportunity to dynamically provide (or take away) additional resources to overloaded (or underloaded) sections of the component pipeline, thus improving the performance of the appliance as a whole.
- **Non-disruptive upgrades:** Often, one needs to upgrade the hardware or software of enterprise systems with little or no disruption in service. The different software components of an appliance can be migrated across physical machines through transparent VM migration, thereby enabling non-disruptive hardware upgrades. The mechanisms that enable higher availability can be leveraged for non-disruptive software upgrades.

Such benefits have prompted enterprise-appliance makers to include virtualization technologies in their systems. The IBM DS8000 series storage system [7] is an example of an appliance that incorporates a hypervisor, albeit in a limited fashion, to host two virtual fault-isolated and performance-isolated storage systems on the same physical hardware. Separation of production and test environments, and flexibility of resource allocation are cited as reasons for incorporating virtualization [7].

2.3 Performance issues with virtualization

Encapsulating the software components of an appliance in VMs introduces new performance issues. First, device access may be considerably slower in a virtualized environment. Second, data transfer between components that used to happen via inexpensive function calls

now crosses protected VM boundaries; since such data transfer is critical to overall performance, it is important that the inter-VM communication between the component VMs be optimized. The first issue is often easily solved in appliances, as devices can be dedicated to components. We address the second performance issue in this paper.

2.4 Inter-VM communication mechanisms

Current inter-VM communication mechanisms rely on either copying (XenLoop [26], XenSocket [28]) or page mapping/unmapping (Netfront [1]) techniques. Both of these techniques incur performance overheads in the critical data path, making them unsuitable for data-traffic intensive server appliances like storage systems. Moreover, the data throughput and latency results obtained with these mechanisms do not satisfy the requirements of an appliance. From another perspective, some of these mechanisms [26, 28] are designed for a specific kind of data traffic - network packets. In addition, they do not offer the flexibility of layering other types of data traffic on top of them. Thereby, restricting the applicability of their solution between different kinds of components in an appliance. All these reasons made us conclude that we need a specialized high-performance inter-VM communication mechanism. Moreover, since multiple component VMs process data in a pipeline fashion, it is not sufficient to have efficient pair wise inter-VM communication; we require efficient end-to-end transitive inter-VM communication.

3 Design and Implementation

In this section, we first describe the design goals of Fido, followed by the inherent trust model that forms the key enabler of our communication mechanism. We then present Fido, our fast inter-VM communication mechanism. Finally, we describe MMNet and MMBlk, the networking and disk access interfaces that build on the communication abstraction provided by Fido.

3.1 Design Goals

The following are the design goals of Fido to enable greater applicability as well as ease of use:

- **High Performance:** Fido should enable high throughput, low latency communication with acceptable CPU consumption.
- **Dynamically Pluggable:** Introduction or removal of Fido should not require a reboot of the system. This enables component VMs to leverage Fido without entailing an interruption in service.
- **Non-intrusive:** In order to limit the exposure of kernel data structures Fido should be built in a non-intrusive fashion. The fewer the dependencies with other kernel data structures, the easier it is to port

across kernel versions.

- **Application-level transparent:** Leveraging Fido should not require applications to change. This ensures that existing applications can start enjoying the performance benefits of Fido without requiring code-level changes.
- **Flexible:** Fido should enable different types of data transfer mechanisms to be layered on top of it with minimal dependencies and a clean interface.

Specifically, being non-intrusive, dynamically pluggable and application transparent extends Fido’s applicability in appliances where the components might be independently developed by collaborating organizations.

3.2 Relaxed Trust Model

Enterprise-class server appliances consist of various software components that are either mostly built by a single organization or put together from pre-tested and qualified components. As a result, the degree of trust between components is significantly more than in typical applications of virtualization. In fact, the various components collaborate extensively and readily exchange or release resources for use by other components. At the same time, in spite of best efforts, the various components may contain bugs that create a need for isolating them from each other.

In an enterprise server appliance, the following trust assumptions apply. First, the different software components in VMs are assumed to be non-malicious. Therefore, read-only access to each other’s address spaces is acceptable. Second, most bugs and corruptions are assumed to lead to crashes sooner than later; enterprise appliances are typically designed to fail-fast; as well, it has been shown that Linux systems often crash within 10 cycles of fault manifestation [10]. Therefore, the likelihood of corruptions propagating from a faulty VM to a communicating VM via read-only access of memory is low. However, VMs are necessary to isolate components from crashes in each other.

3.3 Fido

Fido is an inter-VM shared-memory-based communication mechanism that leverages the relaxed trust model to improve data transfer speed. In particular, we design Fido with a goal of reducing the primary contributors to inter-VM communication overheads: hypervisor transitions and data copies. In fact, Fido enables zero-copy data transfer across multiple virtual machines on the same physical system.

Like other inter-VM communication mechanisms that leverage shared memory, Fido consists of the following features: (i) a shared-memory mapping mechanism, (ii) a signaling mechanism for cross-VM synchronization, and (iii) a connection-handling mechanism that fa-

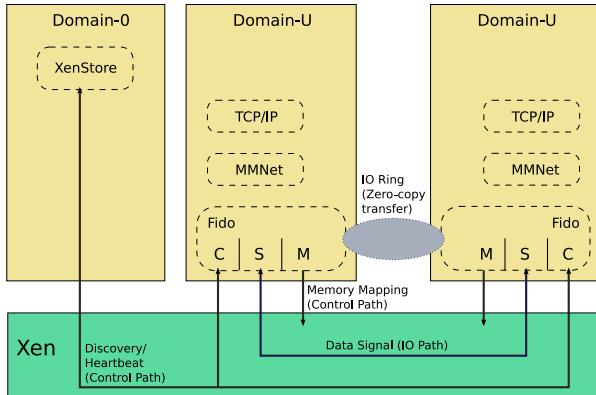


Figure 1: Fido Architecture. The figure shows the components of Fido in a Linux VM over Xen. The two domUs contain the collaborating software components. In this case, they use MMNet and Fido to communicate. Fido consists of three primary components: a memory-mapping module (*M*), a connection module (*C*), and a signaling module (*S*). The connection module uses XenStore (a centralized key-value store in dom0) to discover other Fido-enabled VMs, maintain its own membership, and track VM failures. The memory-mapping module uses Xen grant reference hypervisor calls to enable read-only memory mapping across VMs. It also performs zero-copy data transfer with a communicating VM using I/O rings. The signaling module informs communicating VMs about availability and use of data through the Xen signal infrastructure.

cilitates set-up, teardown, and maintenance of shared-memory state. Implementation of these features requires the use of specific para-virtualized hypervisor calls. As outlined in the following subsections, the functionality expected from these API calls is simple and is available in most hypervisors (Xen, VMWare ESX, etc.).

Fido improves performance through simple changes to the shared-memory mapping mechanism as compared to traditional inter-VM communication systems. These changes are complemented by corresponding changes to connection handling, especially for dealing with virtual-machine failures. Figure 1 shows the architecture of Fido. We have implemented Fido for a Linux VM on top of the Xen hypervisor. However, from a design perspective, we do not depend on any Xen-specific features; Fido can be easily ported to other hypervisors. We now describe the specific features of Fido.

3.3.1 Memory Mapping

In the context of enterprise-class appliance component VMs, Fido can exploit the following key trends: (i) the virtual machines are not malicious to each other and hence each VM can be allowed read-only access to the entire address space of the communicating VM and (ii) most systems today use 64-bit addressing, but individual virtual machines have little need for as big an address space due to limitations on physical memory size. Therefore, with Fido, the entire address space of a *source*

virtual machine is mapped read-only into the *destination* virtual machine, where source and destination refer to the direction of data transfer. This mapping is established *a priori*, before any data transfer is initiated. As a result, the data transfer is limited only by the total physical memory allocated to the source virtual machine, thus avoiding limits to throughput scaling due to small shared-memory segments. Other systems [9, 26] suffer from these limits, thereby causing either expensive hypervisor calls and page table updates [20] or data copies to and from the shared segment when the data is not produced in the shared-memory segment [17, 28].

In order to implement this memory mapping technique, we have used the grant reference functionality provided by the Xen hypervisor. In VMWare ESX, the functional equivalent would be the hypervisor calls leveraged by the VMCI (Virtual Machine Communication Interface [24]) module. To provide memory mapping, we have not modified any guest VM (Linux) kernel data structures. Thus, we achieve one of our design goals of being non-intrusive to the guest kernel.

3.3.2 Signaling Mechanism

Like other shared-memory based implementations, Fido needs a mechanism to send signals between communicating entities to notify data availability. Typically, hypervisors (Xen, VMWare, etc.) support hypervisor calls that enable asynchronous notification between VMs. Fido adopts the Xen signaling mechanism [9] for this purpose. This mechanism amortizes the cost of signaling by collecting several data transfer operations and then issuing one signaling call for all operations. Again, this bunching together of several operations is easier with Fido since the shared memory segment is not limited. Moreover, after adding a bunch of data transfer operations, the source VM signals the destination VM only when it has picked up the previous signal from the source VM. In case the destination VM has not picked up the previous signal, it is assumed that it would pick up the newly queued operations while processing the previously enqueued ones.

3.3.3 Connection Handling

Connection handling includes connection establishment, connection monitoring and connection error handling between peer VMs.

Connection State: A Fido connection between a pair of VMs consists of a shared memory segment (*meta-data segment*) and a Xen event channel for signaling between the VMs. The metadata segment contains shared data structures to implement producer-consumer rings (*I/O rings*) to facilitate exchanging of data between VMs (similar to Xen I/O rings [1]).

Connection Establishment: In order to establish an

inter-VM connection between two VMs, the Fido module in each VM is initially given the identity (Virtual Machine ID - *vmid*) of the peer VM. One of the communicating VMs (for example, the one with the lower *vmid*) initiates the connection establishment process. This involves creating and sharing a metadata segment with the peer. Fido requires a centralized key-value DB that facilitates proper synchronization between the VMs during the connection setup phase. Operations on the DB are not performance critical, they are performed only during setup time, over-the-network access to a generic transactional DB would suffice. In Xen, we leverage XenStore—a centralized hierarchical DB in Dom0—for transferring information about metadata segment pages via an asynchronous, non-blocking handshake mechanism. Since Fido leverages a centralized DB to exchange metadata segment information, it enables communicating VMs to establish connections dynamically. Therefore, by design, Fido is made *dynamically pluggable*.

From an implementation perspective, Fido is implemented as a loadable kernel module, and the communication with XenStore happens at the time of loading the kernel module. Once the metadata segment has been established between the VMs using XenStore, we use the I/O rings in the segment to bootstrap memory-mapping. This technique avoids the more heavy-weight and circuitous XenStore path for mapping the rest of the memory read-only. The source VM’s memory is mapped into the paged region of the destination VM in order to facilitate zero-copy data transfer to devices (since devices do not interact with data in non-paged memory). To create such a mapping in a paged region, the destination VM needs corresponding page structures. We therefore pass the appropriate kernel argument *mem* at boot time to allocate enough page structures for the mappings to be introduced later. Note that Linux’s memory-hotplug feature allows dynamic creation of page structures, thus avoiding the need for a boot-time argument; however, this feature is not fully-functional in Xen para-virtualized Linux kernels.

Connection Monitoring: The Fido module periodically does a heartbeat check with all the VMs to which it is connected. We again leverage XenStore for this heartbeat functionality. If any of the connected VMs is missing, the connection failure handling process is triggered.

Connection Failure Handling: Fido reports errors detected during the heartbeat check to higher-level layers. Upon a VM’s failure, its memory pages that are mapped by the communicating VMs cannot be deallocated until all the communicating VMs have explicitly unmapped those pages. This ensures that after a VM’s failure, the immediate accesses done by a communicating VM will not result in access violations. Fortunately, this is guar-

anteed by Xen’s inter-VM page sharing mechanism.

Data Transfer: This subsection describes how higher layer subsystems can use Fido to achieve zero-copy data transfer.

- **Data Representation:** Data transferred over the Fido connection is represented as an array of pointers, referred to as the scatter-gather (SG) list. Each I/O ring entry contains a pointer to an SG list in the physical memory of the source VM and a count of entries in the SG list. The SG list points to data buffers allocated in the memory of the source VM.
- **IO Path:** In the send data path, every request originated from a higher layer subsystem (i.e., a client of Fido) in the source guest OS is expected to be in an SG list and sent to the Fido layer. The SG list is sent to the destination guest OS over the I/O ring. In the receive path, the SG list will be picked up by the Fido layer and passed up to the appropriate higher layer subsystem, which in turn will package it into a request suitable for delivery to the destination OS. Effectively, the SG list is the generic data structure that enables different higher layer protocols to interact with Fido without compromising the zero-copy advantage.
- **Pointer Swizzling:** A source VM’s memory pages are mapped at an arbitrary offset in the kernel address space of the destination VM. As a result, the pointer to the SG list and the data pointers in the SG list provided by the source VM are incomprehensible when used as-is by the destination VM. They need to be translated relative to the offset where the VM memory is mapped in. While the translation can be done either by the sender or the receiver, we chose to do it in the sender. Doing the translation in the sender simplifies the design of transitive zero-copy (Section 3.3.4).

3.3.4 Transitive Zero-Copy

As explained in Section 2, data flows through an enterprise-class software architecture successively across the different components in a pipeline. To ensure high performance we need true end-to-end zero-copy. In Section 3.3.1, we discussed how to achieve zero-copy between two VMs. In this section, we address the challenges involved in extending the zero-copy transitively across multiple component VMs.

Translation problems with transitive zero-copy: In order to achieve end-to-end zero-copy, data originating in a *source component VM* must be accessible and comprehensible in downstream component VMs. We ensure accessibility of data by mapping the memory of the source component VM in every downstream component VM with read permissions. For data to be comprehensi-

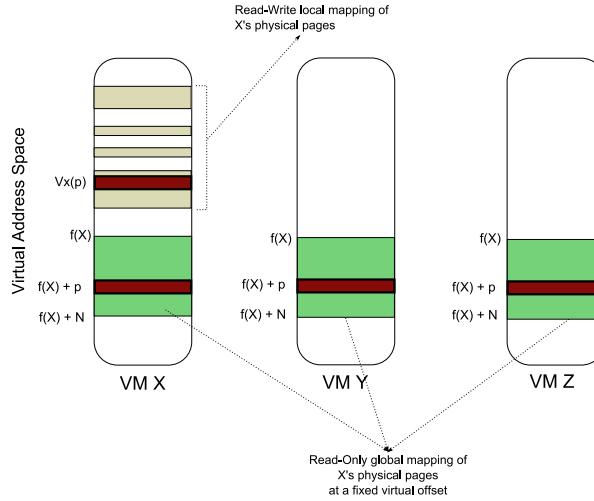


Figure 2: PGVAS technique between VMs X, Y and Z.

ble in a downstream component VM, all data references that are resolvable in the source VM’s virtual address space will have to be translated correctly in the destination VM’s address space. Doing this translation in each downstream VM can prove expensive.

Pseudo Global Virtual Address Space: The advent of 64-bit processor architecture makes it feasible to have a global virtual address space [3, 11] across all component VMs. As a result, all data references generated by a source VM will be comprehensible in all downstream VMs; thus eliminating all address translations.

The global address space systems (like Opal[3]) have a single shared page table across all protected address spaces. Modifying the traditional guest OS kernels to use such a single shared page table is a gargantuan undertaking. We observe that we can achieve the effect of a global virtual address space if each VM’s virtual address space ranges are distinct and disjoint from each other. Incorporating such a scheme may also require intrusive changes to the guest OS kernel. For example, Linux will have to be modified to map its kernel at arbitrary virtual address offsets, rather than from a known fixed offset.

We develop a hybrid approach called *Pseudo Global Virtual Address Space* (PGVAS) that enables us to leverage the benefits of a global virtual address space without the need for intrusive changes to the guest OS kernel. We assume that the virtual address spaces in the participating VMs are 64-bit virtual address spaces; thus the kernel virtual address space should have sufficient space to map the physical memory of a large number of co-located VMs. Figure 2 illustrates the PGVAS technique. With PGVAS, there are two kinds of virtual address mappings in a VM, say X. *Local mapping* refers to the traditional way of mapping the physical pages of X by its guest OS, starting from virtual address zero. In addition, there is a *global mapping* of the physical pages of X at a virtual

offset derived from X’s id, say $f(X)$. An identical global mapping exists at the same offset in the virtual address spaces of all communicating VMs. In our design, we assume VM ids are monotonically increasing, leading to $f(X) = M \cdot X + \text{base}$, where M is the maximum size of a VM’s memory, X is X’s id and base is the fixed starting offset in the virtual address spaces.

To illustrate the benefits, consider a transitive data transfer scenario starting from VM X, leading to VM Y and eventually to VM Z. Let us assume that the transferred data contains a pointer to a data item located at physical address p in X. This pointer will typically be a virtual reference, say $V_x(p)$, in the local mapping of X, and thus, incomprehensible in Y and Z. Before transferring the data to Y, X will encode p to a virtual reference, $f(X) + p$, in the global mapping. Since global mappings are identical in all VMs, Y and Z can dereference the pointer directly, saving the cost of multiple translations and avoiding the loss of transparency of data access in Y and Z. As a result, all data references have to be translated once by the source VM based on the single unique offset where its memory will be mapped in the virtual address space of every other VM. This is also the rationale for having the sender VM do the translations of references in Fido as explained in Section 3.3.1.

3.4 MMNet

MMNet connects two VMs at the network link layer. It exports a standard network device interface to the guest OS. In this respect, MMNet is very similar to Xen Net-Back/NetFront drivers. However, it is layered over Fido and has been designed with the key goal of preserving the zero-copy advantage that Fido provides.

MMNet exports all of the key Fido design goals to higher-layer software. Since MMNet is designed as a network device driver, it uses clean and well-defined interfaces provided by the kernel, ensuring that MMNet is totally *non-intrusive* to the rest of the kernel. MMNet is implemented as a loadable kernel module. During loading of the module, after the MMNet interface is created, a route entry is added in the routing table to route packets destined to the communicating VM via the MMNet interface. Packets originating from applications dynamically start using MMNet/Fido to communicate with their peers in other VMs, satisfying the *dynamic pluggability* requirement. This seamless transition is completely *transparent to the applications* requiring no application-level restarts or modifications.

MMNet has to package the Linux network packet data structure `skb` into the OS-agnostic data-structures of Fido and vice-versa, in a zero-copy fashion. The `skb` structure allows for data to be represented in a linear data buffer and in the form of a non-linear scatter-gather list of buffers. Starting with this data, we create a Fido-

compatible SG list (Section 3.3.3) containing pointers to the `skb` data. Fido ensures that this data is transmitted to the communicating VM via the producer-consumer I/O rings in the metadata segment.

On the receive path, an asynchronous signal triggers Fido to pull the SG list and pass it to the corresponding MMNet module. The MMNet module in turn allocates a new `skb` structure with a custom destructor function and adds the packet data from the SG onto the non-linear part of the `skb` without requiring a copy. Once the data is copied from kernel buffers onto the user-space, the destructor function on the `skb` is triggered. The `skb` destructor function removes the data pointers from the non-linear list of the `skb` and requests Fido to notify the source VM regarding completion of packet data usage.

Though MMNet appears as a network device, it is not constrained by certain hardware limitations like the MTU of traditional network devices and can perform optimizations in this regard. MMNet presents an MTU of 64KB (maximum TCP segment size) to enable high performance network communication. In addition, since MMNet is used exclusively within a physical machine, MMNet can optionally disable checksumming by higher protocol layers, thereby reducing network processing costs.

3.5 MMBlk

MMBlk implements block level connection between virtual machines. Conceptually MMBlk is similar to Xen’s BlkBack/BlkFront block device driver [1]. However, like MMNet, it is layered on top of the Fido

We implement MMBlk as a split block device driver for the Linux kernel. In accordance to a block device interface, MMBlk receives read and write requests from the kernel in the `bio` structure. `bio` provides a description of read/write operations to be done by the device along with an array of pages containing data.

MMBlk write path can be trivially implemented with no modifications to the Linux code. Communicating VMs share their memory in a read-only manner. Thus, a writer VM only needs to send pointers to the `bio` pages containing write data. Then, the communicating VM on the other end can either access written data or in the case of a device driver VM, it can perform a DMA straight from the writer’s pages. Note, that in order to perform DMA, the `bio` page has to be accessible by the DMA engine. This comes with no additional data copy on a hardware providing an IOMMU. An IOMMU enables secure access to devices by enabling use of virtual addresses by VMs. Without an IOMMU, we rely on the `swiotlb` Xen mechanism implementing IOMMU translation in software. `swiotlb` keeps a pool of low memory pages, which are used for DMA. When translation is needed, `swiotlb` copies data into this pool.

Unfortunately, implementation of a zero-copy read

path is not possible without intrusive changes to the Linux storage subsystem. The problem arises from the fact that on the read path, pages into which data has to be read are allocated by the reader, i.e., by an upper layer, which creates the `bio` structure before passing it to the block device driver. These pages are available read-only to the block device driver domain and hence cannot be written into directly. There are at least three ways to handle this problem without violating fault-isolation between the domains. First, the driver VM can allocate a new set of pages to do the read from the disk and later pass it to the reader domain as part of the response to the read request. The reader then has to copy the data from these pages to the original destination, incurring copy costs in the critical path. The second option is to make an intrusive change to the Linux storage subsystem whereby the `bio` structure used for the read contains an extra level of indirection, i.e., pointers to pointers of the original buffers. Once the read data is received in freshly allocated pages from the driver VM, the appropriate pointers can be fixed to ensure that data is transferred in a zero-copy fashion. The third option is similar to the first one, instead of copying we can perform page-flipping to achieve the same goal. We performed a microbenchmark to compare the performance of copying versus page-flipping and observed that page-flipping outperforms copying for larger data transfers (greater than 4K bytes). We chose the first option for our implementation, experimenting with page-flipping is part of future work.

4 Evaluation

In this section, we evaluate the performance of MMNet and MMBlk mechanisms with industry-standard microbenchmarks.

4.1 System Configuration

Our experiments are performed on a machine equipped with two quad-core 2.1 GHz AMD Opteron processors, 16 GB of RAM, three NVidia SATA controllers and two NVidia 1 Gbps NICs. The machine is configured with three additional (besides the root disk) Hitachi Deskstar E7K500 500GB SATA disks with a 16 MB buffer, 7200 RPM and a sustained data transfer rate of 64.8 MB/s. We use a 64-bit Xen hypervisor (version 3.2) and a 64-bit Linux kernel (version 2.6.18.8).

4.2 MMNet Evaluation

We use the `netperf` benchmark (version 2.4.4) to evaluate MMNet. `netperf` is a simple client-server based user-space application, which includes tests for measuring uni-direction bandwidth (STREAM tests) and end-to-end latency (RR tests) over TCP and UDP.

We compare MMNet with three other implementations: i) *Loop*: the loopback network driver in a sin-

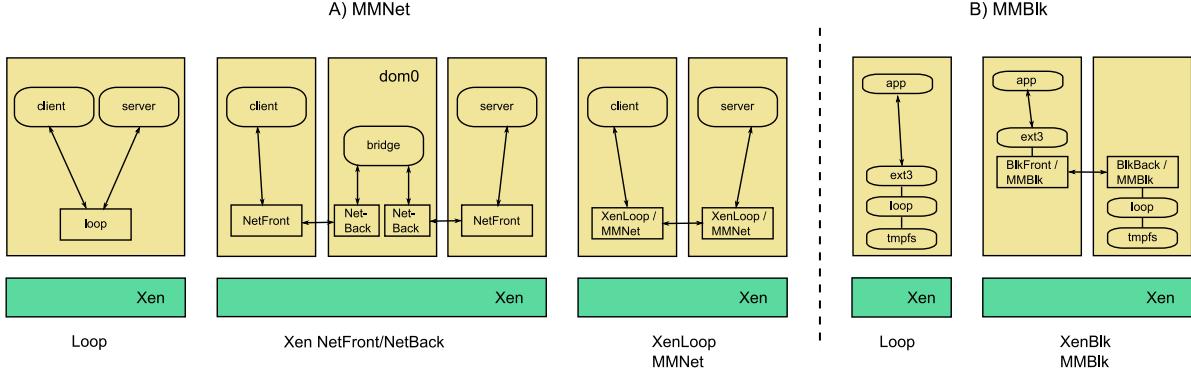


Figure 3: MMNet and MMBlk Evaluation Configurations

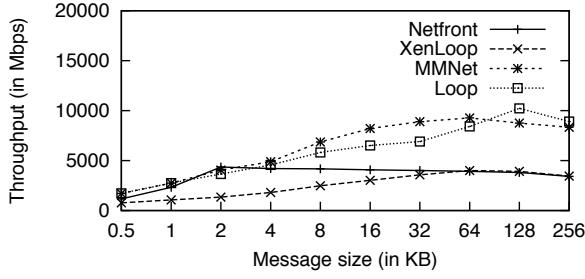


Figure 4: TCP Throughput (TCP_STREAM test)

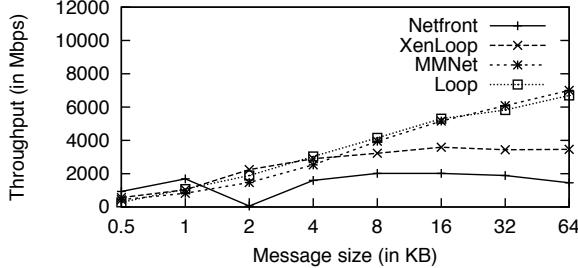


Figure 5: UDP Throughput (UDP_STREAM test)

gle VM for baseline; ii) *Netfront*: the default Xen networking mechanism that routes all traffic between two co-located VMs (*domUs*) through a third management VM (*dom0*), which includes a backend network driver; iii) *XenLoop* [26]: an inter-VM communication mechanism that, like MMNet, achieves direct communication between two co-located *domUs* without going through *dom0*. These configurations are shown in the Figure 3A.

Unlike MMNet, the other implementations have additional copy or page remapping overheads in the I/O path, as described below:

- **Netfront:** In the path from the sender *domU* to *dom0*, *dom0* temporarily maps the sender *domU*'s pages. In the path from *dom0* to the receiver *domU*, either a copy or *page-flipping* [1] is performed. In our tests we use *page-flipping*, which is the default mode.

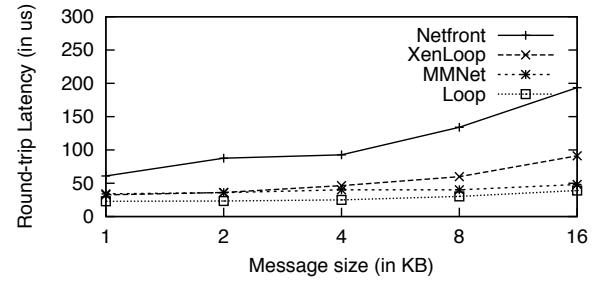


Figure 6: TCP Latency (TCP_RR test)

- **XenLoop:** A fixed region of memory is shared between the two communicating *domUs*. In the I/O path, XenLoop copies data in and out of the shared region.

All VMs are configured with one virtual CPU each. The only exception is the VM in the *loop* experiment, which is configured with two virtual CPUs. Virtual CPUs were pinned to separate physical cores, all on the same quad-core CPU socket. All reported numbers are averages from three trials.

Figure 4 presents TCP throughput results for varying message sizes. The figure shows that MMNet performs significantly better than XenLoop and the default Xen drivers, reaching a peak throughput of 9558 Mb/s at a message size of 64KB.

We see that performance with XenLoop is worse than Netfront. Given that XenLoop was designed to be more efficient than Netfront, this result seems contradictory. We found that the results reported by the XenLoop authors [26] were from tests performed on a single socket, dual-core machine. The three VMs, namely the two *domUs* and *dom0*, were sharing two processor cores amongst themselves. In contrast, our tests had dedicated cores for the VMs. This reduces the number of VM switches and helps Netfront better pipeline activity (such as copies and page-flips) over three VMs. In order to verify this hypothesis, we repeated the netperf TCP_STREAM experiment (with a 16KB message size)

by restricting all the three VMs to two CPU cores and found that XenLoop (4000 Mbps) outperforms Netfront (2500 Mbps).

UDP throughput results for varying message sizes are shown in Figure 5. We see that the MMNet performance is very similar to Loop and significantly better than Netfront and XenLoop. Inter-core cache hits could be the reason for this observation, since UDP protocol processing times are shorter compared to TCP, it could lead to better inter-core cache hits. This will benefit data copies done across cores (for example, in XenLoop, the receiver VM’s copy from the shared region to the kernel buffer will be benefited). There will be no benefit for Netfront because it does page remapping as explained earlier.

Figure 6 presents the TCP latency results for varying request sizes. MMNet is almost four times better than Netfront. Moreover, MMNet latencies are comparable to XenLoop for smaller message sizes. However, as the message sizes increase, the additional copies that XenLoop incurs hurt latency and hence, MMNet outperforms XenLoop. Netfront has the worst latency results because of the additional dom0 hop in the network path.

4.3 MMBlk Evaluation

We compare the throughput and latency of MMBlk driver with two other block driver implementations: i) *Loop*: the monolithic block layer implementation where the components share a single kernel space; ii) *XenBlk*: a split architecture where the block layer spans two VMs connected via the default Xen block device drivers. These configurations are illustrated in Figure 3B.

To eliminate the disk bottleneck, we create a block device (using *loop* driver) on TMPFS. In the Loop setup, an ext3 file system is directly created on this device. In the other setups, the block device is created in one (*backend*) VM and exported via the XenBlk/MMBlk mechanisms to another (*frontend*) VM. The frontend VM creates an ext3 file system on the exported block device. The backend and the frontend VMs were configured with 4 GB and 1 GB of memory, respectively. The in-memory block device is 3 GB in size and we use a 2.6 GB file in all tests.

Figure 7 presents the memory read and write throughput results for different block sizes measured using the IOZone [15] microbenchmark (version 3.303). For the Loop tests, we observe that the IOZone workload performs poorly. To investigate this issue, we profiled executions of all three setups. Compared to the split cases, execution of Loop has larger number of wait cycles. From our profile traces, we believe that the two filesystems (TMPFS and ext3) compete for memory – trying to allocate new pages. TMPFS is blocked as most of the memory is occupied by the buffer cache staging ext3’s writes. To improve Loop’s performance, we configure the monolithic system with 8GB of memory.

We consistently find that read throughput at a particular record size is better than the corresponding write throughput. This is due to soft page faults in TMPFS for new writes (writes to previously unwritten blocks).

From Figure 7A, we see that MMBlk writes perform better than XenBlk writes by 39%. This is because XenBlk incurs page remapping costs in the write path, while MMBlk does not. Further, due to inefficiencies in Loop, on average MMBlk is faster by 45%. In the case of reads, as shown in Figure 7B, XenBlk is only 0.4% slower than the monolithic Loop case. On smaller record sizes, Loop outperforms XenBlk due to a cheaper local calls. On larger record sizes, XenBlk becomes faster leveraging the potential to batch requests and better pipeline execution. XenBlk outperforms MMBlk by 35%. In the read path, MMBlk does an additional copy, whereas XenBlk does page remapping. Eliminating the copy (or page flip) in the MMBlk read path is part of future work.

5 Case Study: Virtualized Storage System Architecture

Commercial storage systems [8, 14, 21] are an important class of enterprise server appliances. In this case study, we examine inter-VM communication overheads in a virtualized storage-system architecture and explore the use of Fido to alleviate these overheads. We first describe the architecture of a typical network-attached storage system, then we outline a proposal to virtualize its architecture and finally, evaluate the performance of the virtualized architecture.

5.1 Storage System Architecture

The composition of the software stack of a storage system is highly vendor-specific. For our analysis, we use the NetApp software stack [27] as the reference system. Since all storage systems need to satisfy certain common customer requirements and have similar components and interfaces, we believe our analysis and insights are also applicable to other storage systems in the same class as our reference system.

The data flow in a typical monolithic storage system is structured as a pipeline of requests through a series of components. Network packets are received by the network component (e.g., network device driver). These packets are passed up the network stack for protocol processing (e.g., TCP/IP followed by NFS). The request is then transformed into a file system operation. The file system, in turn, translates the request into disk accesses and issues them to a software-RAID component. RAID converts the disk accesses it receives into one or more disk accesses (data and parity) to be issued to a storage component. The storage component, in turn, performs the actual disk operations. Once the data has been retrieved from or written to the disks, an appropriate re-

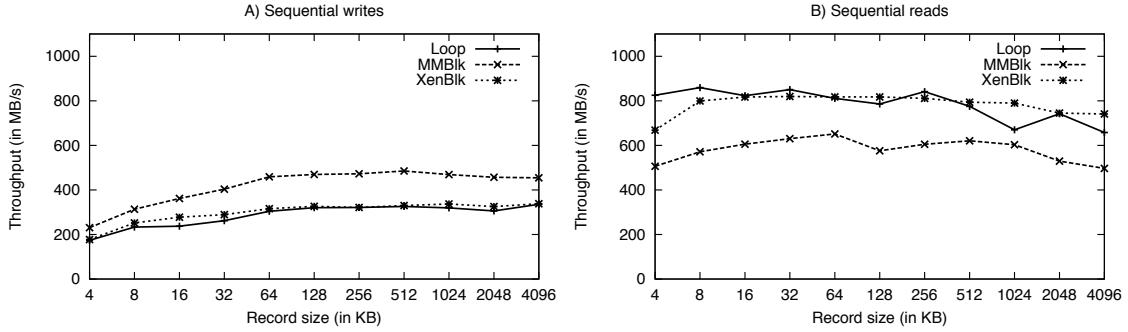


Figure 7: MMBlk Throughput Results

sponse is sent via the same path in reverse.

5.2 Virtualized Architecture

We design and implement a modular architecture for an enterprise-class storage system that leverages virtualization technologies. Software components are partitioned into separate VMs. For the purposes of understanding the impact of inter-VM communication in such an architecture as well as evaluating our mechanisms, we partition components as shown in Figure 8. While this architecture is a representative architecture, it might not necessarily be the ideal one from a modularization perspective. Identifying the ideal modular architecture merits a separate study and is outside the scope of our work.

Our architecture consists of four different component VMs — Network VM, Protocols and File system VM, RAID VM and Storage VM. Such an architecture can leverage many benefits from virtualization (Section 2.2):

- Virtualization provides much-needed fault isolation between components. In addition, the ability to reboot individual components independently greatly enhances the availability of the appliance.
- Significant performance isolation across file system volumes can be achieved by having multiple sets of File system, RAID, and Storage VMs, each set serving a different volume. One can also migrate one such set of VMs to a different physical machine for balancing load.
- Component independence helps with faster development and deployment of new features. For instance, changes to device drivers in the Storage VM (say to support new devices or fix bugs) can be deployed independently of other VMs. In fact, one might be able to upgrade components in a running system.

The data flow in the virtualized architecture starts from the Network VM, passes successively through the File system and RAID VMs and ends in the Storage VM, resembling a pipeline. This pipelined processing requires data to traverse several VM boundaries entailing inter-VM communication performance overheads. In order to ensure high end-to-end performance of the system, it

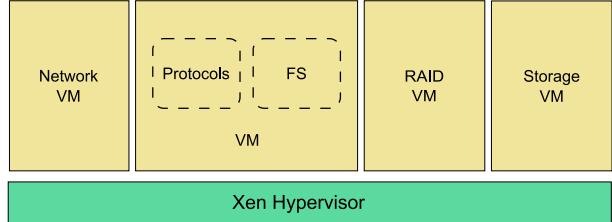


Figure 8: Architecture with storage components in VMs

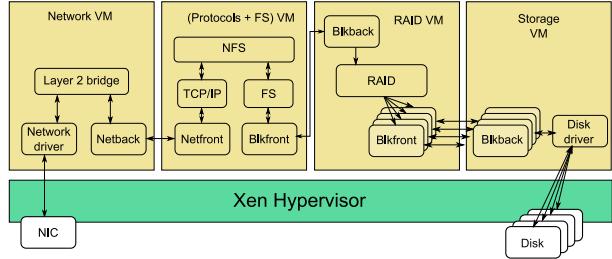


Figure 9: Full system with MMNet and MMBlk

is imperative that we address the inter-VM communication performance. As mentioned in Section 2, inter-VM communication performance is just one of the performance issues for this architecture; other issues like high-performance device access are outside our scope.

5.3 System Implementation Overview

In Figure 9, we illustrate our full system implementation incorporating MMNet and MMBlk between the different components. Our prototype has four component VMs:

- **Network VM:** The network VM has access to the physical network interface. In addition, it has a MMNet network device to interface with the (Protocols + FS) VM. The two network interfaces are linked together by means of a Layer 2 software bridge.
- **Protocols + FS VM:** This VM is connected to the Network VM via the MMNet network device. We run an in-kernel NFSv3 server exporting an ext3 file system to network clients via the MMNet interface. The file system is laid out on a RAID device exported by the RAID VM via an MMBlk block device. This

VM is referred to as FS VM subsequently.

- **RAID VM:** This VM exports a RAID device to the FS VM using the MMBlk block device interface. We use the MD software RAID5 implementation available in Linux. The constituting data and parity disks are actually virtual disk devices exported again via MMBlk devices from the neighboring storage VM.
- **Storage VM:** The storage VM accesses physical disks, which are exported to the RAID VM via separate MMBlk block device interfaces.

The goal of our prototype is to evaluate the performance overheads in a virtualized storage system architecture. Therefore, we did not attempt to improve the performance of the base storage system components themselves by making intrusive changes. For example, the Linux implementation of the NFS server incurs two data copies in the critical path. The first copy is from the network buffers to the NFS buffers. This is followed by another copy from the NFS buffer to the Linux buffer cache. The implication of these data copies is that we cannot illustrate true end-to-end transitive zero-copy. Nevertheless, for a subset of communicating component VMs, i.e., from the FS VM onto the RAID and Storage VMs, transitive zero-copy is achieved by incorporating our PGVAS enhancements (Section 3.3.4). This improves our end-to-end performance significantly.

5.4 Case Study Evaluation

To evaluate the performance of the virtualized storage system architecture, we run a set of experiments on the following three systems:

- **Monolithic-Linux:** Traditional Linux running on hardware, with all storage components located in the same kernel address space.
- **Native-Xen:** Virtualized storage architecture with four VMs (Section 5.3) connected using the native Xen inter-VM communication mechanisms—Netfront/NetBack and Blkfront/Blkback.
- **MM-Xen:** Virtualized storage architecture with MMNet and MMBlk as shown in Figure 9.

Rephrasing the evaluation goals in the context of these systems, we expect that for the MMNet and MMBlk mechanisms to be *effective*, performance of MM-Xen should be significantly better than Native-Xen and for the virtualized architecture to be *viable*, the performance difference between Monolithic-Linux and MM-Xen should be minimal.

5.4.1 System Configuration

We now present the configuration details of the system. The physical machine described in Section 4.1 is used as the storage server. The client machine, running Linux (kernel version 2.6.18), has similar configuration as the

server, except for the following differences: two dual-core 2.1 GHz AMD Opteron processors, 8 GB of memory and a single internal disk. The two machines are connected via a Gigabit Ethernet switch.

In the Monolithic-Linux experiments, we run native Linux with eight physical cores and 7 GB of memory. In the Native-Xen and MM-Xen experiments, there are four VMs on the server (the disk driver VM is basically dom0). The FS VM is configured with two virtual CPUs, each of the other VMs have one virtual CPU. Each virtual CPU is assigned to a dedicated physical processor core. The FS VM is configured with 4 GB of memory and the other VMs are configured with 1 GB each. The RAID VM includes a Linux MD [22] software RAID5 device of 480 GB capacity, constructed with two data disks and one parity disk. The RAID device is configured with a 1024 KB chunk size and a 64 KB stripe cache (write-through cache of recently accessed stripes). The ext3 file system created on the RAID5 device is exported by the NFS server in “async” mode. The “async” export option mimics the behavior of enterprise-class networked storage systems, which typically employ some form of non-volatile memory to hold dirty write data [12] for improved write performance. Finally, the client machine uses the native Linux NFS client to mount the exported file system. The NFS client uses TCP as the transport protocol with a 32 KB block size for read and write.

5.4.2 Microbenchmarks

We use the IOZone [15] benchmark to compare the performance of Monolithic-Linux, Native-Xen and MM-Xen. We perform read and write tests, in both sequential and random modes. In each of these tests, we vary the IOZone record sizes, but keep the file size constant. The file size is 8 GB for both sequential and random tests.

Figure 10 presents the throughput results. For sequential writes, as shown in Figure 10A, MM-Xen achieves an average improvement of 88% over the Native-Xen configuration. This shows that Fido performance improvements help the throughput of data transfer significantly. Moreover, MM-Xen outperforms even Monolithic-Linux by 9.5% on average. From Figure 10C, we see that MM-Xen achieves similar relative performance even with random writes. This could be due to the benefits of increased parallelism and pipelining achieved by running VMs on isolated cores. In the monolithic case, kernel locking and scheduling inefficiencies could limit such pipelining. Even with sequential reads, as shown in Figure 10B, MM-Xen outperforms both Monolithic-Linux and Native-Xen by about 13%. These results imply that our architecture has secondary performance benefits when the kernels in individual VMs exhibit SMP inefficiencies.

With random workloads, since the size of the test file

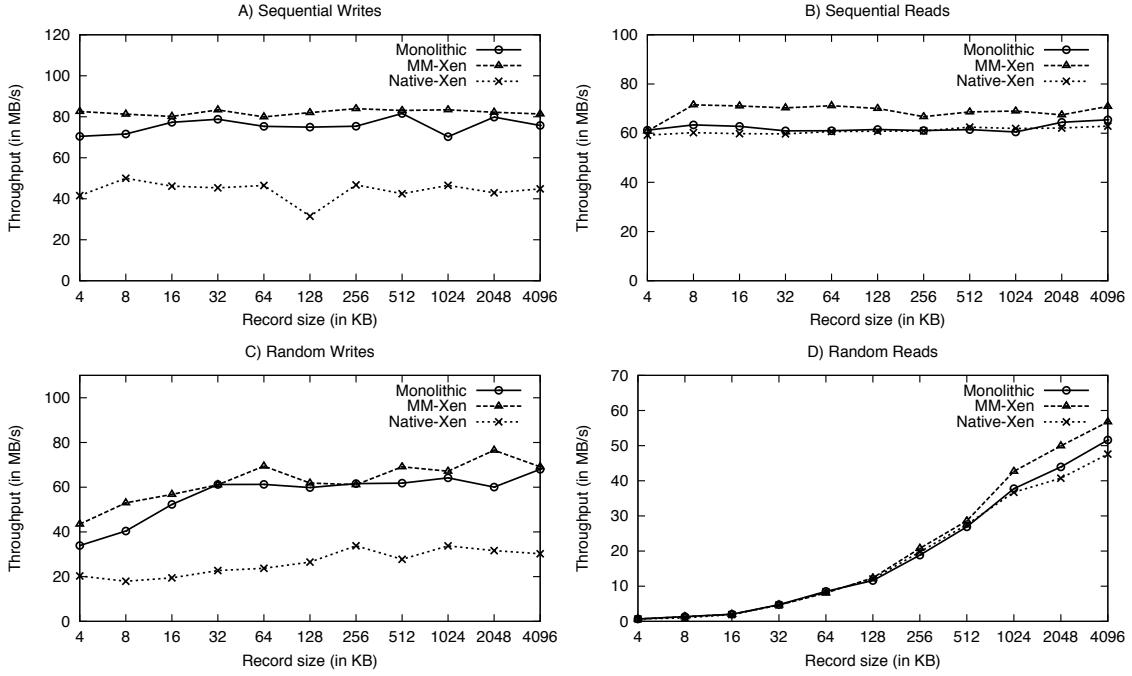


Figure 10: IOZone Throughput Results

remains constant, the number of seeks reduces as we increase the record size. This explains why the random read throughput (Figure 10D) increases with increasing record sizes. However, random writes (Figure 10C) do not exhibit similar throughput increase due to the mitigation of seeks by the coalescing of writes in the buffer cache (recall that the NFS server exports the file system in "async" mode).

Finally, Figure 11 presents the IOZone latency results. We observe that MM-Xen is always better than Native-Xen. Moreover, MM-Xen latencies are comparable to Monolithic-Linux in all cases.

5.4.3 Macrobenchmarks

TPCC-UVa [18] is an open source implementation of the TPC-C benchmark version 5. TPC-C simulates read-only and update intensive transactions, which are typical of complex OLTP (On-Line Transaction Processing) systems. TPCC-UVa is configured to run a one hour test, using 50 warehouses, a ramp-up period of 20 minutes and no database vacuum (garbage collection and analysis) operations.

Table 1 provides a comparison of TPC-C performance across three configurations: Monolithic-Linux, Native-Xen, and MM-Xen. The main TPC-C metric is tpmC, the cumulative number of transactions executed per minute. Compared to Monolithic-Linux, Native-Xen exhibits a 38% drop in tpmC. In contrast, MM-Xen is only 3.1% worse than Monolithic-Linux.

The response time numbers presented in Table 1 are averages of the response times from five types of transac-

	tpmC (transactions/min)	Avg. Response Time (sec)
Monolithic	293.833	26.5
Native-Xen	183.032	350.8
MM-Xen	284.832	30.4

Table 1: TPC-C Benchmark Results

tions that TPC-C reports. We see that MM-Xen is within 13% of the average response time of Monolithic-Linux. These results demonstrate that our inter-VM communication improvements in the form of MMNet and MMBlk translate to good performance with macrobenchmarks.

6 Related Work

In this section we first present a survey of the different existing inter-VM communication approaches and articulate the trade-offs between them. Subsequently, since we use a shared-memory communication method, we articulate how our research leverages and complements prior work in this area.

6.1 Inter-VM Communication Mechanisms

Numerous inter-VM communication mechanisms already exist. Xen VMM supports a restricted inter-VM communication path in the form of Xen split drivers [9]. This mechanism incurs prohibitive overheads due to data copies or page-flipping via hypervisor calls in the critical path. XenSocket [28] provides a socket-like interface. However, XenSocket approach is not transparent. That is, the existing socket interface calls have to be changed. XenLoop [26] achieves efficient inter-VM communica-

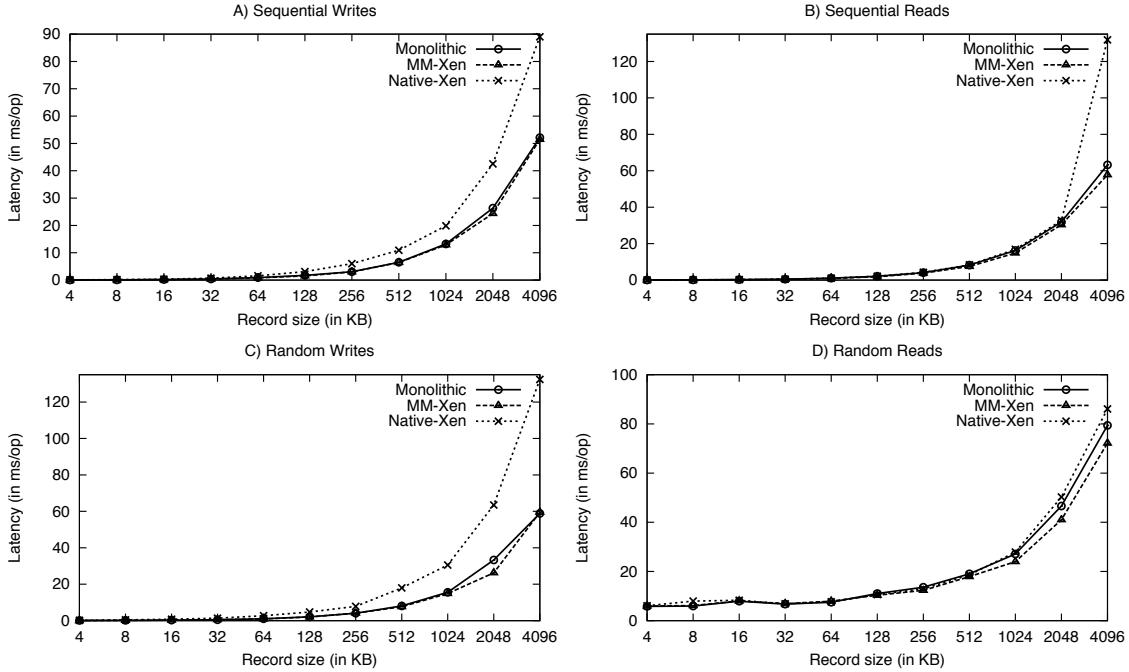


Figure 11: IOZone Latency Results

tion by snooping on every packet and short-circuiting packets destined to co-located VMs. While this approach is transparent, as well as non-intrusive, its performance trails MMNet performance since it incurs copies due to a bounded shared memory region between the communicating VMs. The XWay [17] communication mechanism hooks in at the transport layer. Moreover, this intrusive approach is limited to applications that are TCP oriented. In comparison to XWay and XenSocket, MMNet does not require any change in the application code, and MMNet’s performance is better than XenLoop and XenSocket. Finally, IVC [13] and VMWare VMCI [24] provide library level solutions that are not system-wide.

6.2 Prior IPC Research

A lot of prior research has been conducted in the area of inter-process communication. Message passing and shared-memory abstractions are the two major forms of IPC techniques. Mechanisms used in Fbufs [6], IO-Lite [23], Beltway buffers [5] and Linux Splice [19] are similar to the IPC mechanism presented in this paper.

Fbufs is an operating system facility for I/O buffer management and efficient data transfer across protection domains on shared memory machines. Fbufs combine virtual page remapping and memory sharing. Fbufs target throughput of I/O intensive applications that require significant amount of data to be transferred across protection boundaries. A buffer is allocated by the sender with appropriate write permissions whereas the rest of the I/O paths access it in read-only mode. Thus, buffers are immutable. However, append operation is supported by ag-

gregating multiple data-buffers into a logical message. Fbufs employ the following optimizations: a) mapping of buffers into the same virtual address space (removes lookup for a free virtual address) b) buffer reuse (buffer stays mapped in all address spaces along the path) and c) allows volatile buffers (sender doesn’t have to make them read-only upon send). IO-Lite is similar in spirit to Fbufs, it focuses on zero-copy transfers between kernel modules by means of unified buffering. Some of the design principles behind Fbufs and IO-Lite can be leveraged on top of PGVAS in a virtualized architecture.

Beltway buffers [5] trade protection for performance implementing a zero-copy communication. Beltway allocates a system-wide communication buffer and translates pointers to them across address spaces. Beltway does not describe how it handles buffer memory exhaustion except for the networking case, in which it suggests to drop packets. Beltway enforces protection per-buffer, making a compromise between sharing entire address spaces and full isolation. Compared to us, Beltway simplifies pointer translation across address spaces – it translates only a pointer to buffer, inside the buffer linear addressing is used, so indexes inside the buffer remain valid across address spaces.

splice [19] is a Linux system call providing a zero-copy I/O path between processes (i.e. a process can send data to another process without lifting them to user-space). Essentially, Splice is an interface to access the in-kernel buffer with data. This means that a process can forward the data but cannot access it in a zero-copy way. Buffer memory management is implemented

through reference counting. Splice "copy" is essentially a creation of a reference counted pointer. Splice appeared in Linux since 2.6.17 onwards.

7 Conclusion

In this paper, we present Fido, a high-performance inter-VM communication mechanism tailored to software architectures of enterprise-class server appliances. On top of Fido, we have built two device abstractions-MMNet and MMBlk exporting the performance characteristics of Fido to higher layers. We evaluated MMNet and MMBlk separately as well in the context of a virtualized network-attached storage system architecture and we observe almost imperceptible performance penalty due to these mechanisms. In all, employing Fido in appliance architectures makes it viable for them to leverage virtualization technologies.

References

- [1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03*, New York, 2003.
- [2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — A technique for cheap recovery. In *OSDI'04*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.
- [3] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Ladowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, 1994.
- [4] Cisco Systems. Cisco Products. <http://www.cisco.com/products>.
- [5] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the os traffic jam. In *Proceedings of INFOCOM 2008*, 2008.
- [6] P. Druschel and L. L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 189–202, New York, NY, USA, 1993. ACM.
- [7] B. Dufrasne, W. Gardt, J. Jamsek, P. Kimmel, J. Myyryläinen, M. Oscheka, G. Pieper, S. West, A. Westphal, and R. Wolf. IBM System Storage DS8000 Series: Architecture and Implementation, Apr. 2008.
- [8] EMC. The EMC Celerra Family. <http://www.emc.com/products/family/celerra-family.htm>.
- [9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *OASIS*, Oct 2004.
- [10] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of Linux Kernel Behavior under Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, 2003.
- [11] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Softw. Pract. Exper.*, 28(9):901–928, 1998.
- [12] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.
- [13] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda. Virtual Machine Aware Communication Libraries for High Performance Computing. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007*, Reno, Nevada, USA, November 2007.
- [14] IBM Corporation. IBM Storage Controllers. <http://www-03.ibm.com/systems/storage/network/index.html>.
- [15] IOZone. IOZone Filesystem Benchmark. <http://www.iozone.org>.
- [16] Juniper Networks. Juniper Networks Products. <http://www.juniper.com/products>.
- [17] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, New York, NY, USA, 2008. ACM.
- [18] D. R. Llanos. Tpcc-uva: an open-source tpc-c implementation for global performance measurement of computer systems. *SIGMOD Rec.*, 35(4):6–15, 2006.
- [19] L. McVoy. The splice I/O model. <http://ftp.tux.org/pub/sites/ftp.bitmover.com/pub/splice.ps>.
- [20] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.
- [21] NetApp, Inc. NetApp Storage Systems. <http://www.netapp.com/products>.
- [22] OSDL. Overview - Linux-RAID. <http://linux-raid.osdl.org/index.php/Overview>.
- [23] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *ACM Transactions on Computer Systems*, pages 15–28, 2000.
- [24] VMWare. Virtual Machine Communication Interface. http://pubs.vmware.com/vmci-sdk/VMCI_intro.html.
- [25] VMWare. VMWare Inc. <http://www.vmware.com>.
- [26] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: A Transparent High Performance Inter-VM Network Loopback. In *Proc. of International Symposium on High Performance Distributed Computing (HPDC)*, June 2008.
- [27] A. Watson, P. Benn, A. G. Yoder, and H. Sun. Multi-protocol Data Access: NFS, CIFS, and HTTP. Technical Report 3014, NetApp, Inc., Sept. 2001.
- [28] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Middleware 2007: ACM/IFIP/USENIX 8th International Middleware Conference*, Newport Beach, CA, USA, November 2007.

STOW: A Spatially and Temporally Optimized Write Caching Algorithm

Binny S. Gill

IBM Almaden Research Center

Biplob Debnath

University of Minnesota

Michael Ko

IBM Almaden Research Center

Wendy Belluomini

IBM Almaden Research Center

Abstract

Non-volatile write-back caches enable storage controllers to provide quick write response times by hiding the latency of the disks. Managing a write cache well is critical to the performance of storage controllers. Over two decades, various algorithms have been proposed, including the most popular, LRW, CSCAN, and WOW. While LRW leverages temporal locality in the workload, and CSCAN creates spatial locality in the destages, WOW combines the benefits of both temporal and spatial localities in a unified ordering for destages. However, there remains an equally important aspect of write caching to be considered, namely, the rate of destages. For the best performance, it is important to destage at a steady rate while making sure that the write cache is not under-utilized or over-committed. Most algorithms have not seriously considered this problem, and as a consequence, forgo a significant portion of the performance gains that can be achieved.

We propose a simple and adaptive algorithm, STOW, which not only exploits both spatial and temporal localities in a new order of destages, but also facilitates and controls the rate of destages effectively. Further, STOW partitions the write cache into a sequential queue and a random queue, and dynamically and continuously adapts their relative sizes. Treating the two kinds of writes separately provides for better destage rate control, resistance to one-time sequential requests polluting the cache, and a workload-responsive write caching policy.

STOW represents a leap ahead of all previously proposed write cache management algorithms. As anecdotal evidence, with a write cache of 32K pages, serving a 4+P RAID-5 array, using an SPC-1 Like Benchmark, STOW outperforms WOW by 70%, CSCAN by 96%, and LRW by 39%, in terms of measured throughput. STOW consistently provides much higher throughputs coupled with lower response times across a wide range of cache sizes, workloads, and experimental configurations.

1 Introduction

In spite of the recent slowdown in processor frequency scaling due to power and density issues, the advent of multi-core technology has enabled processors to continue their relentless increase in I/O rate to storage systems. In contrast, the electro-mechanical disks have not been able to keep up with a comparable improvement in access times. As this schism between disk and processor speeds widens, caching is attracting significant interest.

Enterprise storage controllers use caching as a fundamental technique to hide I/O latency. This is done by using fast but relatively expensive random access memory to hold data belonging to slow but relatively inexpensive disks.

Over a period of four decades, a large number of read cache management algorithms have been devised, including Least Recently Used (LRU), Frequency-Based Replacement (FBR) [19], Least Frequently Recently Used (LFRU) [15], Low Inter-Reference Recency Set (LIRS) [13], Multi-Queue (MQ) [24], Adaptive Replacement Cache (ARC) [17], CLOCK with Adaptive Replacement (CAR) [2], Sequential Prefetching in Adaptive Replacement Cache (SARC) [8], etc. While, the concept of a write cache has been around for over two decades, we realize that it is a more complex and less studied problem. We focus this paper on furthering our understanding of write caches and improving significantly on the state of the art.

1.1 What Makes a Good Write Caching Algorithm?

A write-back (or fast-write) cache relies on fast, non-volatile storage to hide latency of disk writes. It can contribute to performance in five ways. It can (i) harness temporal locality, thereby reducing the number of pages that have to be destaged to disks; (ii) leverage spatial locality, by reordering the destages in the most

disk-friendly order, thereby reducing the average cost of destages; (iii) absorb write bursts from applications by maintaining a steady and reasonable amount of free space, thereby guaranteeing a low response time for writes; (iv) distribute the write load evenly over time to minimize the impact to concurrent reads; and (v) serve read hits that occur within the write cache.

There are two critical decisions regarding destaging in write caching: the *destage order* and the *destage rate*. The destage order deals with leveraging temporal and spatial localities, while the destage rate deals with guaranteeing free space and destaging at a smooth rate. Write caching has so far been treated mainly as an eviction problem, with most algorithms focusing only on the destage order. The most powerful write caching algorithms will arise when we explore the class of algorithms that simultaneously optimize for both the destage order and the destage rate.

1.2 Our Contributions

Firstly, we present a detailed analysis of the problem of managing the destage rate in a write cache. While this has remained a relatively unexplored area of research, we demonstrate that it is an extremely important aspect of write caching with the potential of significant gains if done right. Further, we show that to manage the destage rate well, we actually need a new destaging order.

Secondly, we present a Spatially and Temporally Optimized Write caching algorithm (STOW), that for the first time, exploits not only temporal and spatial localities, but also manages both the destage rate and destage order effectively in a single powerful algorithm that handsomely beats popular algorithms like WOW, CSCAN, and LRW, across a wide range of experimental scenarios. Anecdotally, with a write cache of 32K pages (and high destage thresholds), serving a RAID-5 array, the measured throughput for STOW at 20 ms response time, outperform WOW by 70%, CSCAN by 96%, and LRW by 39%. STOW consistently and significantly outperforms all other algorithms across a wide range of cache sizes, workload intensities, destage threshold choices, and backend RAID configurations.

1.3 Outline of the paper

In Section 2, we briefly survey previous related research. In Section 3, we explore why the destage rate is a crucial aspect of any good write caching algorithm. In Section 4, we present the new algorithm STOW. In Section 5, we describe the experimental setup and workloads, and in Section 6, we present our main quantitative results. Finally, in Section 7 we conclude with the main findings of this paper.

2 Related Work

Although an extensive amount of work has been done in the area of read caching, not all techniques are directly applicable to write caching. While read caching is essentially a two dimensional optimization problem (maximizing hit ratio and minimizing prefetch wastage), write caching is a five dimensional optimization problem (as explained in Section 1.1).

In a write-back cache, the response time for a write is small if there is space in the write cache to store the new data. The data in the write cache is destaged to disks periodically, indirectly affecting any concurrent reads by increasing their average service response time. To reduce the number of destages from the write cache to the disks, it is important to leverage temporal locality and, just like in read caches, maximize the hit (overwrite) ratio. The primary way to maximize temporal locality is to attempt to evict the least recently written (LRW) pages from the cache. An efficient approximation of this is available in the CLOCK [5] algorithm which is widely used in operating systems and databases. These algorithms, however, do not account for the spatial locality factor in write cache performance.

Orthogonally, the order of destages can be chosen so as to minimize the average cost of each destage. This is achieved by destaging data that are physically proximate on the disks. Such spatial locality maximization has been studied mostly in the context of disk scheduling algorithms, such as shortest seek time first (SSTF) [6], SCAN [6], cyclical SCAN (CSCAN) [20], LOOK [18], VSCAN [7], and FSCAN [4]. Some of these require knowledge of the current state of the disk head [12, 21], which is either not available or too cumbersome to track in the larger context of storage controllers. Others, such as CSCAN, order the destages by logical block address (LBA) and do not rely on knowing the internal state of the disk.

In the first attempt to combine spatial and temporal locality in write caching for storage systems [10], a combination of LRW [1, 3, 11] and LST [10, 22] was used to balance spatial and temporal locality. This work had the drawback that it only dealt with one disk and it did not adapt to the workload.

In general, the order of destages is different for leveraging temporal locality versus spatial locality. One notable write caching algorithm, Wise Ordering for Writes (WOW) [9], removed this apparent contradiction, by combining the strength of CLOCK [5] in exploiting temporal locality and Cyclical SCAN (CSCAN) [20] in exploiting spatial locality. As shown in Figure 1, WOW groups the pages in the cache in terms of write groups and sorts them in their LBA order. To remember if a write group was recently used, a recency bit is main-

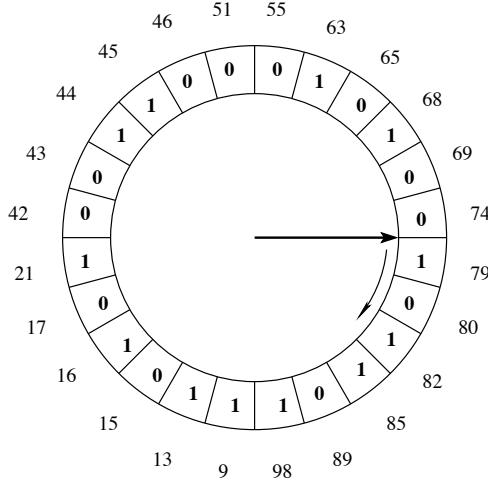


Figure 1: The data structure of the WOW algorithm

tained in each write group. When a page is written in a write group already present in the cache, the recency bit of the write group is set to 1. Destages are done in the sorted order of the write groups. However, if the recency bit of a write group is 1, the write group is bypassed after resetting the recency bit to 0.

While WOW solves the *destage order* problem, the *destage rate* problem has attracted little research. Both WOW and an earlier work on destage algorithms [23] use a linear thresholding scheme that grows the rate of destages linearly as the number of modified pages in the cache grows. While this scheme is quite robust, destage orders like WOW and CSCAN cannot achieve their full potential due to a destructive interaction between the destage rate and the destage order policies. We are unaware of any research that studies the interaction between the two vital aspects of write caching: the destage order and the destage rate.

	CSCAN	LRW	WOW	STOW
Spatial Locality	Yes	No	Yes	Yes
Temporal Locality	No	Yes	Yes	Yes
Scan Resistance	No	No	Little	Yes
Stable Destage Rate	No	Little	No	Yes
Stable Occupancy	No	Little	No	Yes

Table 1: Comparison of Various Write Cache Algorithms

Table 1 shows how the set of algorithms discussed above compare. LRW considers only recency (temporal locality), CSCAN considers only spatial locality, and WOW considers both spatial and temporal locality. Our algorithm, STOW (Spatially and Temporally Optimized Writes), tracks spatial and temporal localities, and is *scan resistant* because it shields useful random data from being pushed out due to the influx of large amounts se-

quential data. STOW also avoids large fluctuations in the destage rate and cache occupancy.

3 Taming the Destage Rate

Historically caching has always been treated as an eviction problem. While it might be true for demand-paging read caches, it is only partially true for write caches. The rate of eviction or the destage rate has attracted little research so far. In this section we explore why the destage rate is a crucial aspect of any good write caching algorithm.

3.1 The Goals

Any good write caching algorithm needs to manage the destage rate to achieve the following three objectives: (i) Match the destage rate (if possible) to the average incoming rate to avoid hitting 100% full cache condition (leading to synchronous writes); (ii) Avoid underutilizing the write cache space; (iii) Destage smoothly to minimally impact concurrent reads.

3.2 Tutorial: How to Get it Wrong?

Rather than simply present our approach, we explain why we reject other seemingly reasonable approaches, some of which have been used in the past.

3.2.1 Ignore Parity Groups

More often than not, a write cache in a storage controller serves a RAID array involving parity groups (e.g. RAID-5, RAID-6). In such scenarios, it is important to group together destages of separate pages within the same parity group to minimize the number of parity updates. The best case happens when all members of the parity group are present in the cache. The parity group can optionally be extended to beyond one parity stripe or even to RAID-10 (see WOW [9]).

3.2.2 Destage Quickly

One approach is to destage as soon as there are dirty pages and as fast as the system would allow. While this would guarantee that the cache stays away from the full-cache condition most effectively even for strong workloads, it wipes out any temporal and spatial locality benefits for gentler workloads. The left panel in Figure 2 shows that a quick destaging policy can lead to very low cache occupancy.

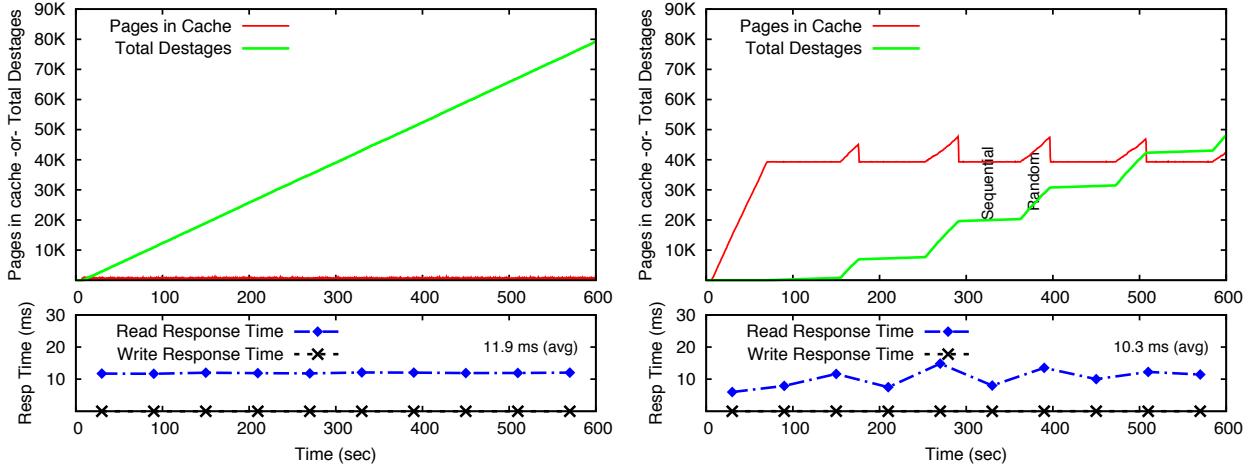


Figure 2: Left panel: We destage in the order specified by the WOW algorithm, and as fast as the disks allow. The cache remains relatively empty and the observed read response time is around 11.9 ms on average. Right panel: We destage in the WOW order, however we destage only if the cache is more than 60% full (out of 64K pages). We observe a better average read response time, but the cache occupancy exhibits spikes when random data is destaged. When sequential data is destaged, the disks are under-utilized as shown by the flat steps in the total destages curve.

3.2.3 Fixed Threshold

In the right panel of Figure 2, we examine a policy which destages quickly only if the cache is more than 60% full. This performs better on average but displays ominous “spikes” in cache occupancy which correspond to spikes in the read response times. The non-uniform destage rate is bad for concurrent reads. Furthermore, a higher fixed threshold is more likely to hit 100% cache occupancy, while a lower fixed threshold underutilizes the write cache most of the time.

The Spikes: Any write caching algorithm destages writes in a particular order. If the workload is such that the algorithm destages sequential data for some time followed by random data, such peaks are inevitable because destaging random data is far more time consuming and during such intervals, the cache occupancy can spike even for steady workloads. With WOW or CSCAN, such spikes appear when the workload has a sequential and a random component that target different portions of the LBA space. This is commonly observed in both real-life and benchmark workloads.

3.2.4 Linear Threshold

In the WOW [9] paper, a linear threshold scheme is proposed which is better than the fixed threshold scheme because it provides a gradual gradation of destage rates which is more friendly to concurrent reads and also allows the thresholds to be safely closer to the 100% mark. For example, instead of destaging at full force after reaching 60% occupancy, linear threshold would use a number of concurrent destage requests that is propor-

tional to how close the write cache is to a high threshold (say 80% occupancy). Beyond the high threshold it will destage at the full rate. This is the best scheme so far that we are aware of, however, it cannot address the spike problem, as evidenced in Figure 3. In this paper we use an H/L notation for the thresholds: e.g. 90/80 implies that the high threshold is at 90% of the cache and the low threshold is at 80%.

3.2.5 Adaptive Threshold

One might suggest that we should develop a scheme that adaptively determines the correct high and low thresholds for the linear threshold scheme. However, the reader will observe that the spikes are very tall, and any such effort will result in a serious underutilization of the write cache space.

3.3 Maybe Not WOW Then?

While the order of destages proposed in the WOW algorithm is disk-friendly, the same order makes the destage rate problem a tough nut to crack. We propose to change the destage order to allow us to tame the destage rate.

3.3.1 Separate Random and Sequential Data

The spikes in cache occupancy are caused by long alternating regimes of sequential and random destages. We separate the sequential data and the random data into two WOW-like data-structures in the write cache. Whenever there is a need to destage, we destage from the larger of the two queues, as a first approximation. Later, we shall

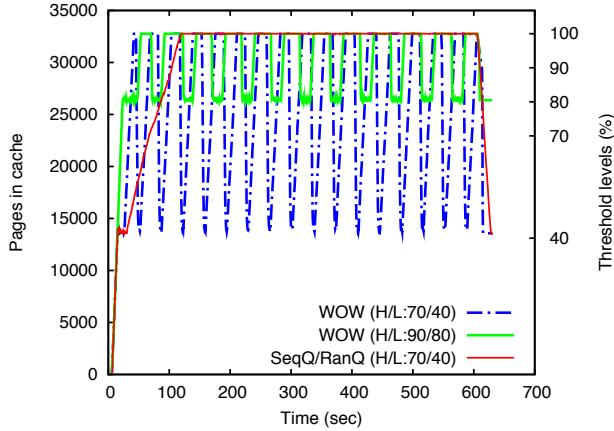


Figure 3: We examine cache occupancy for a workload with both sequential and random writes when using the linear thresholding scheme to determine destage rate. WOW exhibits spikes which go down up to the low threshold when sequential data is destaged, and rise when random data is destaged. For higher thresholds, WOW hits cache full condition more often. If we destage sequential and random data together (SeqQ/RanQ), we eliminate the spikes, but the cache is almost always full.

see that the ideal partitioning of the cache requires adapting to the workload. While the intermixing of destages from the sequential and the random queues eliminates the spikes beautifully (the SeqQ/RanQ curve in Figure 3), it pollutes the spatial locality in the destage order by sending the disk heads potentially to two separate regions on the disk causing longer seek times. The SeqQ/RanQ variant suffers from the full-cache condition almost all the time, nullifying any gains from eliminating the spikes.

3.3.2 Use Hysteresis in Destaging

We need to be able to control the spikes in cache occupancy without derailing the spatial locality of the destages. We discovered that if we destage no less than a fixed hysteresis amount from the larger queue, we reduce the negative impact of having two destage sources. This brings us to the STOW algorithm which integrates all our intuitions so far (and some more) into a powerful, practical and simple write caching algorithm.

4 STOW: The Algorithm

4.1 The STOW Principle

The STOW algorithm uses two WOW-like sorted circular queues for housing random and sequential data separately. The relative sizes of the queues is continuously adapted according to workload to maximize per-

formance. The decisions for: which queue to destage from, when to destage, and at what rate, are carefully managed to leverage spatial and temporal locality in the workload, as well as to maintain steady cache occupancy levels and destage rates. Despite its simplicity, STOW represents the most comprehensive and powerful algorithm for write cache management.

4.2 Data Structures

4.2.1 Honoring Parity Groups

A write hit (over-write) on a page generally implies that the page and its neighbors are likely to be accessed again. In the context of a storage controller connected to a RAID controller, we would like to postpone the destage of such pages hoping to absorb further writes, thereby reducing the destage load on the disks.

In RAID, each participating disk contributes one strip (e.g. 64 KB) towards each RAID stripe. A RAID stripe consists of logically contiguous strips, one from each data disk. Destaging two distinct pages in the same RAID stripe together is easier than destaging them separately, because in the former case the parity strip needs to be updated only once. We say a **stripe hit** has occurred when a new page is written in a RAID stripe that already has a member in the write cache. In RAID 5, therefore, a stripe hit saves two disk operations (the read and write of the parity strip), while a hit on a page saves four.

While we divide the write cache in pages of 4KB each, we manage the data structures in terms of **write groups**, where a write group is defined as a collection of a fixed number (one or more) of logically consecutive RAID stripes. In this paper, we define the write group to be equal to a RAID stripe. We say a write group is present in the cache if at least one of its member pages is physically present in the cache. Managing the cache in terms of such write groups allows us to better exploit both temporal locality by saving repeated destages and spatial locality by issuing writes in the same write group together, thus minimizing parity updates.

4.2.2 Separating Sequential From Random

Sequential data, by definition, has high spatial locality, and appears in large clumps in the sorted LBA space for algorithms such as WOW or CSCAN. In between clumps of sequential data are random data. We discovered that destaging random data, even when it is sorted, is far more time consuming than destaging sequential data. Therefore, when the destage pointer is in an area full of random data, the slower destage rate causes the cache occupancy to go up. These spikes in the cache occupancy could lead to full-cache conditions even for steady workloads, severely impacting the cache performance. Fur-

ther, during a spike, a cache is especially vulnerable to reaching the 100% occupancy mark even with smaller write bursts. This discovery led us to partition the cache directory in STOW into two separate queues RanQ and SeqQ for the “random” and the “sequential” components of a workload. It is easy to determine whether a write is sequential by looking for the presence of earlier pages in the cache [8] and keeping a counter. The first few pages of a sequential stream are treated as random. If a page is deemed to belong to a sequential stream it is populated in SeqQ; otherwise, the page is stored in RanQ.

Separating sequential data from random data is a necessary first step towards alleviating the problem caused by the spikes in the cache occupancy. However, this is not sufficient by itself, as we will learn in Section 4.3.4 when we discuss destaging.

4.2.3 Two Sorted Circular Queues

The STOW cache management policy is depicted in Algorithm 1. In each queue, RanQ or SeqQ, the write groups are arranged in an ascending order of logical block addresses (LBA), forming a circular queue, much like WOW, as shown in Figure 4. A destage pointer (akin to a clock arm) in each queue, points to the next write group in the queue to be considered for destaging to disk. Upon a page miss (a write that is not an over-write), if the write group does not exist in either queue, the write group with the new page is inserted in the correct sorted order in either RanQ or SeqQ depending on whether the page is determined to be a random or a sequential page. If the write group already exists, then the new page is inserted in the existing write group.

The LBA ordering in both queues allows us to minimize the cost of a destage which depends on how far the disk head would have to seek to complete an operation. The write groups, on the other hand, allow us to exploit any spatio-temporal locality in the workload, wherein a write on one page in a write group suggests an imminent write to another page within the same write group. Not only do we save on parity updates, but we also have the opportunity to coalesce consecutive pages together into the same write operation.

When either RanQ or SeqQ is empty, because the workload lacks the random or the sequential component, then STOW converges to WOW, and by the same token, is better than CSCAN and LRU.

4.3 Operation

4.3.1 What to Destage From a Queue?

The destage pointer traverses the sorted circular queue looking for destage victims. Write groups with a recency bit of 1 are skipped after resetting the recency bit to 0.

Algorithm 1 STOW: Cache Management Policy

Write page x in write group g :

```

1: if  $g \in RanQ \cup SeqQ$  then //a write group hit
2:   if  $x \notin RanQ \cup SeqQ$  then //page miss
3:     allocate  $x$  from FreePageQueue
4:     insert  $x$  in  $g$ 
5:   end if
6:   if  $x$  is sequential and is last page of  $g$  then
7:     set recency-bit of  $g$  to 0
8:   else
9:     set recency-bit of  $g$  to 1
10:  end if
11: else
12:   allocate  $g$  from FreeWriteGroupQueue
13:   allocate  $x$  from FreePageQueue
14:   insert  $x$  into  $g$ 
15:   if  $x$  is sequential then
16:     insert  $g$  into sorted queue in  $SeqQ$ 
17:     if  $x$  is last page of  $g$  then
18:       set recency-bit of  $g$  to 0
19:     else
20:       set recency-bit of  $g$  to 1
21:     end if
22:   else
23:     insert  $g$  into sorted queue in  $RanQ$ 
24:     set recency-bit of  $g$  to 0
25:   end if
26: end if

```

If the recency bit of the write group was found to be 0, then the pages present in the write group are destaged. Thus, write groups with a recency bit of one get an extra life equal to the time it takes for the destage pointer to go around the clock once.

Setting the recency bit in RanQ: When a new write group is created in RanQ, the recency bit is set to 0 (line 24 in Algorithm 1). On a subsequent page hit or a write group hit, the recency bit is set to 1 (line 9), giving all present members of the write group a longer life in the cache, during which they can exploit any overwrites of the present pages or accumulate new neighboring pages in the same write group. This leads to enhanced hit ratio, fewer parity updates, and coalesced writes reducing the total number of destages.

Setting the recency bit in SeqQ: Whenever a page is written to SeqQ, the recency bit in the corresponding write group is set to 1 (lines 9 and 20). This is because we anticipate that subsequent pages will soon be written to the write group by the sequential stream. Destaging to disk is more efficient if the whole write group is present since this avoids the extra read-modify-write of the parity group in a RAID-5 configuration and also coalesces consecutive pages into the same disk IO if possi-

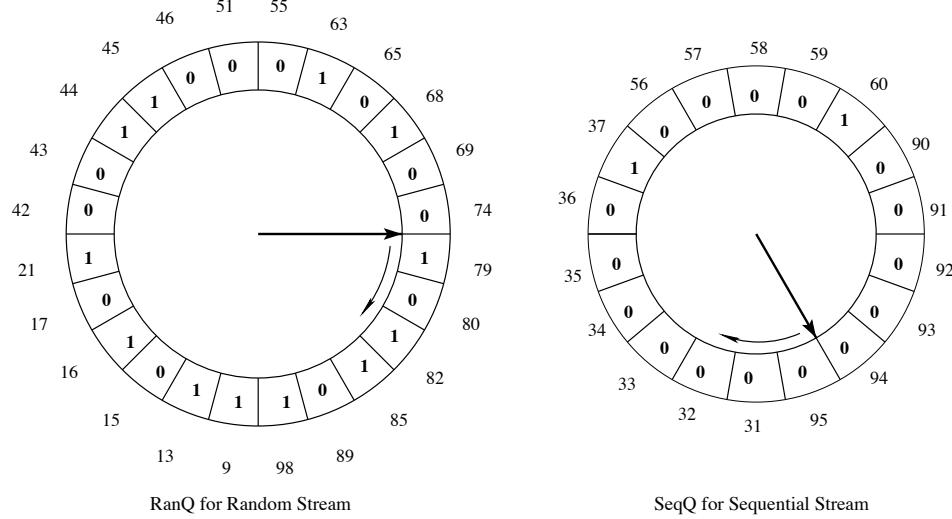


Figure 4: The data structure of the STOW algorithm

ble. However, if the page written is the last page of the write group then the recency bit is forced to 0 (lines 7 and 18). Since the last page of the write group has been written, we do not anticipate any further writes and can free up the cache space the next time the destage pointer visits this write group.

4.3.2 What Rate to Destage at?

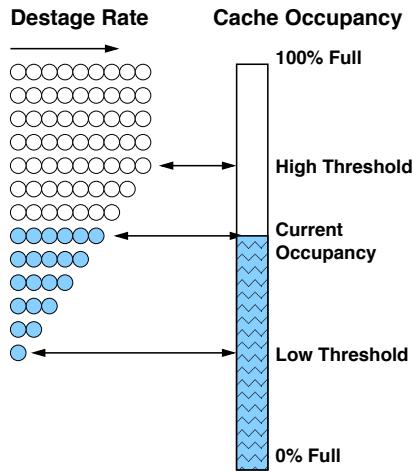


Figure 5: Linear threshold scheme for determining destage rate based on how close the cache occupancy is to the high threshold.

In STOW, we use a linear threshold scheme (see Figure 5, as described in WOW[9]) to determine when and at what rate to destage. We set a low threshold and a high threshold for the cache occupancy. When the cache occupancy is below the low threshold, we leave the write

data in the cache in order to gather potential write hits. When the cache occupancy is above the high threshold, we destage data to disks at the full rate in order to avoid the full-cache condition which is detrimental to response time. When the cache occupancy is between the low and high threshold, the number of concurrent destage requests is linearly proportional to how close we are to the high threshold. Note, a higher number of concurrent destage requests to the disks results in a higher throughput for destages, but of course at the cost of making the disks busier and the concurrent reads slower. The maximum number of concurrent destage requests (queue depth) is set to a reasonable 20 [9] in our experiments.

4.3.3 Which Queue to Destage From?

Algorithm 3 shows how STOW calculates and adapts the desired size of SeqQ (DesiredSeqQSize). Algorithm 2 destages from SeqQ if it is larger than DesiredSeqQSize, else, it destages from RanQ (line 3). While strictly following this simple policy eliminates any deleterious spikes in the cache occupancy, it is not optimal because it sends the disk heads to possibly two distinct locations (the sorted order from RanQ and from SeqQ) simultaneously, resulting in an inter-mingling of two sorted orders, polluting the spatial locality in the destages.

Once we have decided to destage from a queue, we should stick with that decision for a reasonable amount of time, so as to minimize the spatial locality pollution caused by the mixing of two sorted orders. To realize this, we define a fixed number called the **HysteresisCount**. Once a decision has been made to destage from a particular queue, we continue destaging from the same queue until, (i) we have destaged **Hystere-**

Algorithm 2 STOW: Destage Policy

```
1: while needToDestage() do
2:   if hysteresisCountDone() then
3:     if |SeqQ| > DesiredSeqQSize then
4:       set currentDestagePtr to SeqQDestagePtr
5:     else
6:       set currentDestagePtr to RanQDestagePtr
7:     end if
8:   end if
9:   g = write group pointed to by currentDestagePtr
10:  while g->recency-bit == 1 do
11:    g->recency-bit = 0
12:    g = advanceDestagePtr(currentDestagePtr)
13:  end while
14:  destage all pages in g
15:  move destaged pages to FreePageQueue
16:  move g to FreeWriteGroupQueue
17:  advanceDestagePtr(currentDestagePtr)
18: end while
```

sisCount pages from the queue; or (ii) either queue has since grown by more than HysteresisCount pages. Note that destages from RanQ are slower and the second condition avoids a large buildup in SeqQ in the meantime. Once either condition is met, we reevaluate which queue to destage from.

Normally, we fix HysteresisCount to be equal to 128 times the number of spindles in the RAID array. This ensures that a reasonable number of destage operations are performed in one queue's sorted order, before moving to the other queue's sorted order. However, we observed that fluctuations in the cache occupancy are proportional to the HysteresisCount. To maintain a smooth destage rate these fluctuations need to be small relative to the difference between the high and low thresholds. Therefore, we limit HysteresisCount to be no more than 1/8th of the difference (in terms of pages) between the thresholds.

4.3.4 Adapting the Queue Sizes

As we stated earlier, we use the size of SeqQ relative to DesiredSeqQSize for determining which queue to destage from, every time we have destaged HysteresisCount pages. Therefore, we need to wisely and continuously adapt DesiredSeqQSize to be responsive to the workload so as to maximize the aggregate utility of the cache. The marginal utility, in terms of IOPS gained, of increasing the size of either RanQ or SeqQ, is not well understood. Therefore, we propose intuitive heuristics that are very simple to calculate and result in good performance.

Marginal utility for RanQ: We would like to estimate the extra IOs incurred if we make RanQ smaller by unit

Algorithm 3 STOW: Queue Size Management Policy

```
1: if page x in write group g is written then
2:   if g ∈ RanQ then //(RAID-10: use x ∈ RanQ)
3:     if g->recency-bit == 0 then
4:       if (|SeqQ| - DesiredSeqQSize) <
          HysteresisCount then
5:         DesiredSeqQSize -= 1
6:       end if
7:     end if
8:   end if
9: end if
10: if write group g is destaged then
11:   if g not contiguous with previous destage then
12:     if previous stretch < queue depth then
13:       if |RanQ| / (|RanQ| + |SeqQ|) >
          RanRq / (RanRq + SeqRq) then
14:         DesiredSeqQSize += n * |RanQ| / |SeqQ|
15:         //n = num of disks in RAID5 or RAID10
16:       end if
17:     end if
18:   end if
19: end if
```

cache size (a page). We first approximate the number of misses that would be incurred if we reduce the size of RanQ. Let h be the hit rate for first time hits in RanQ (where the recency bit is previously zero). We consider only page hits for RAID-10 but any stripe hit for RAID-5, since in RAID-5, stripe hits save parity updates (two IOs) and are more common than page hits. Assuming a uniform distribution of these hits, we can compute the density of hits to be $h/|RanQ|$. Since a cache does have diminishing returns as its size grows, we add a factor of 0.5 (empirically determined). Each extra miss results in two extra IOs to the disk, yielding a marginal utility of $h/|RanQ|$.

Marginal utility for SeqQ: We would like to estimate the extra IOs incurred by making SeqQ smaller by unit cache size. We first measure the rate, s , at which there are breaks in the logical write group addresses being destaged from SeqQ. Each contiguous group of pages destaged is called a *stretch*. The smaller the size of SeqQ, the higher is the rate s . Since s is inversely proportional to the cache size, the marginal increase in s equal to $s/|SeqQ|$ (since $s*|SeqQ| = \text{const}$, $\frac{ds}{d|SeqQ|} = -\frac{s}{|SeqQ|}$). Each extra break in SeqQ results in one extra write to all n disks. This yields a marginal utility of $n * s/|SeqQ|$.

We adapt the sizes of RanQ and SeqQ targeting a condition where $h/|RanQ| = n * s/|SeqQ|$, to minimize the IOs to the disk, maximizing the performance of the cache.

We implement the above in Algorithm 3 as follows:

Initialization: The initial value of DesiredSeqQSize

is the size of SeqQ when the write cache first reaches the low threshold of destaging.

Decrement: DesiredSeqQSize is reduced by one if: We have a hit (page hit for RAID-10 and write group hit for RAID-5) in RanQ (line 2), where the recency bit is zero (line 3), and the DesiredSeqQSize is not already HysteresisCount lower than the current SeqQ size (line 4).

Increment: DesiredSeqQSize is incremented whenever there is break in the logical addresses of the write groups in SeqQ being destaged (line 11). The amount incremented is $n * |RanQ| / |SeqQ|$, where n is the number of disks in the RAID array. There are two conditions when we do not increment DesiredSeqQSize: (i) When the break in the logical address occurred after a relatively long *stretch* (more than what the queue depth allows to be destaged together) (line 12); or (ii) RanQ is already below its rightful share of the cache based on the proportion of random requests in the workload (line 13).

5 Experimental Setup

A schematic diagram of the experimental system is depicted in Figure 6.

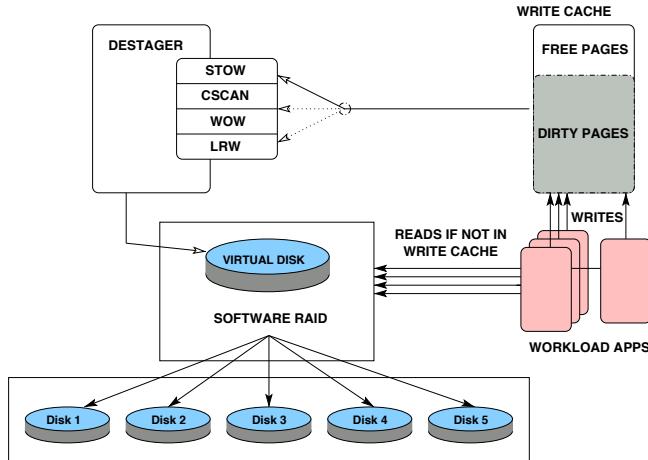


Figure 6: Overall design of the experimental system

5.1 The Basic Hardware Setup

We use an IBM xSeries 3650 machine equipped with two Intel Xeon 3 GHz processors, 4 GB DDR2 memory at 667 MHz, and eight 2.5" 10K RPM SAS disks (IBM 40K1052, 4.5 ms avg. seek time) of 73.4 GB each. A Linux kernel (version 2.6.23) runs on this machine to host all our applications and standard workload generators. We employ five SAS disks for our experiments, and one for the operating system, our software, and workloads.

5.2 Storage Configuration

We study two popular RAID configurations, viz. RAID-5 and RAID-10, using Linux Software RAID. We issue direct I/O to the virtual RAID disk device, always bypassing the kernel buffer. For RAID-5, we use 5 SAS disks to create an array consisting of 4 data disks and 1 parity disk. We choose the stripe size for each disk to be 64 KB, with the resulting stripe group size being 256 KB. For RAID-10, we use 4 SAS disks to create an array in a 2 + 2 configuration. We use the same stripe size of 64 KB for each disk.

We use the entire available storage in one configuration which we call the **Full Backend**. For RAID-5, with the storage capacity of four disks, Full Backend amounts to 573 million 512-byte sectors. For RAID-10, with the storage capacity of two disks, Full Backend amounts to 286 million 512-byte sectors. We also define a **Partial Backend** configuration, where we use only 1/100th of the available storage. While *Full Backend* is characterized by large disk seeks and low hit ratio, the *Partial Backend* generates only short seeks coupled with high hit ratios.

5.3 The Cache

For simplicity, we use volatile DDR2 memory as our write cache. In a real life storage controller, the write cache is necessarily non-volatile (e.g. battery-backed). In our setup, the write cache is managed outside the kernel so that its size can be easily varied allowing us to benchmark a wide range of write cache sizes.

We do not use a separate read cache in our experiments for the following reason. Read misses disrupt the sequentiality of destages determined by any write caching algorithm. A read cache reduces the read misses and amplifies the gains of the better write caching algorithm. Therefore the most adverse environment for a write caching algorithm is when there is no read cache. This maximizes the number of read misses that the disks have to service concurrent to the writes and provides the most valuable comparison of write caching algorithms.

Nevertheless, we do service read hits from the write cache for consistency.

5.4 SPC-1 Benchmark

SPC-1 [16, 14] is the most respected performance benchmark in the storage industry. The benchmark simulates real world environments in a typical server class computer system by presenting a set of I/O operations that are typical for business critical applications like OLTP systems, database systems and mail server applications. We use a prototype implementation of the SPC-1 benchmark that we refer to as SPC-1 Like.

The SPC-1 workload roughly consists of 40% read requests and 60% write requests. For each request, there is a 40% chance that the request is sequential and a 60% chance that the request is random with some temporal locality. SPC-1 scales the footprint of the workload based on the amount of storage space specified. Therefore for a given cache size, the number of read and write hits will be larger if the backend is smaller (*Partial Backend*), and smaller if the amount of storage exposed to the benchmark is larger (*Full Backend*).

SPC-1 assumes three disjoint application storage units (ASU). ASU-1 is assigned 45% of the available back-end storage and represents “Data Store”. ASU-2 is assigned 45% and represents “User Store”. The remaining 10% is assigned to ASU-3 and represents “Log/Sequential Write”. In all configurations, we lay out ASU-3 at the outer rim of the disks followed by ASU-1 and ASU-2.

6 Results

We compare the performance of LRW, CSCAN, WOW and STOW under a wide range of cache size, workload, and backend configurations. We use linear thresholding to determine the rate of destages for all algorithms.

6.1 Stable Occupancy and Destage Rate

6.1.1 Full Backend

In Figure 7(a), we observe that the occupancy graph for WOW as well as CSCAN fluctuates wildly between the low threshold and the 100% occupancy level. For the same scenario, LRW’s cache occupancy remains at 100% occupancy, which implies that most of the time it does not have space for new writes. Only STOW exhibits measured changes in the overall cache occupancy, consistently staying away from the full-cache condition. Note that with linear thresholding, the destage rate is a function of the cache occupancy, and consequently, large fluctuations are detrimental to performance.

The sequential writes in the SPC-1 benchmark are huddled in a small fraction of the address space. As the destage pointer in WOW or CSCAN moves past this sequential region and into the subsequent random region, the occupancy graph spikes upwards because the cache cannot keep up with the incoming rate while destaging in the random region. This disparity can be so large that even the maximum destage concurrency may not be sufficient to keep up with the incoming rate, leading to the dreaded full-cache condition (Figure 7(a)).

Also note the flat bottoms at the low threshold on the occupancy graphs for WOW and CSCAN in Figure 7(a). Since destaging sequential data is quick and easy, the cache occupancy quickly drops down close to the low

threshold, where it uses only a small portion of the allowed destage queue depth to keep up with the incoming rate of the overall workload. The lackadaisical destage rate in the sequential region results in underutilization of disks which is ironic given that the disks cannot keep up with the incoming rate when destages move to the subsequent random regions.

6.1.2 Partial Backend

In Figure 7(b), we observe that the fluctuations in WOW are compressed together. This is because the higher hit ratio causes the destage pointer in WOW to advance much more quickly. CSCAN, on the other hand, does not skip over recently hit pages, and produces less frequent fluctuations but stays in the full-cache condition most of the time.

STOW wisely alternates between the two types of destages, ensuring that the disks are continuously utilized at a relatively constant rate. This eliminates large fluctuations in the occupancy curve for STOW in both the full and partial backend cases.

6.2 Throughput and Response Time

Presenting meaningful results: We use the best way to present results for write caching improvements: the throughput-response time curve. We present gains in terms of bandwidth improvements at the same (reasonable) response time. Another approach is to present gains in terms of response times at the same throughput. E.g., at 900 IOPS in Figure 8(a), we could report at least a 10x improvement in response time over the contenders. While it is accurate, we believe, it is not as informative because it can be cherry-picked to aggrandize even modest gains in such “hockey-stick” plots.

Backward bending: We also observe the *backward bending* phenomenon in some curves which happens whenever a storage controller is overdriven [9]. In this regime, congestion caused by the increasing queue lengths, lock contention, and CPU consumption, bogs down a storage controller such that the disks no longer remain the bottleneck.

6.2.1 Full Backend

In the top panel of Figure 8, we compare the average response time (aggregate read and write) as a function of the throughput achieved when the target throughput is gradually increased in the SPC1-Like workload generator. Overall, STOW outperforms all algorithms significantly across all load levels. Observe that WOW and CSCAN improve as the threshold range becomes wider since a wider range allow them to contain the fluctuations better, hitting the full-cache condition for lesser

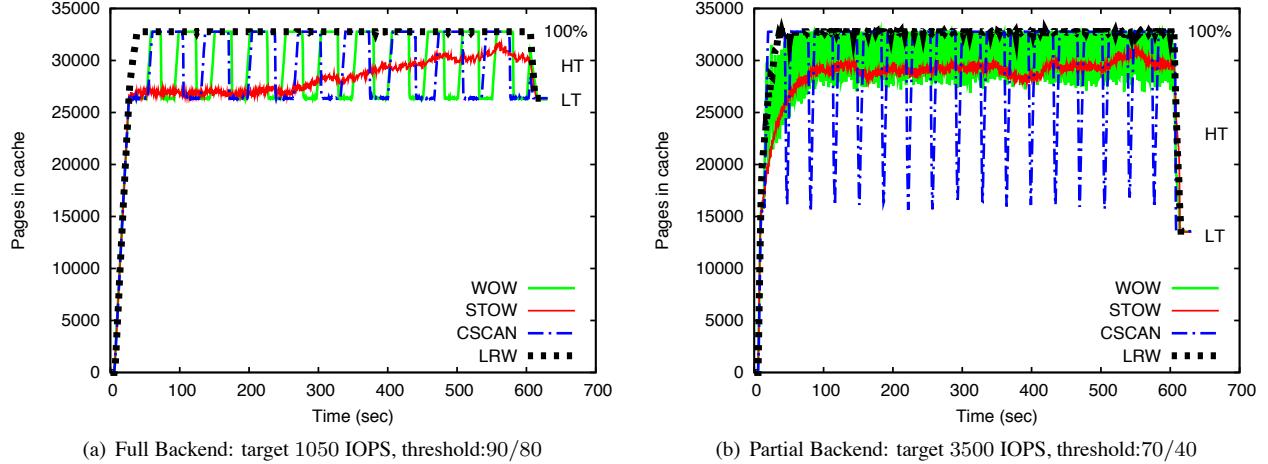


Figure 7: Cache occupancy as a function of time in a 32K page cache serving RAID-5 (RAID-10 is similar). STOW neither exhibits large fluctuations in cache occupancy, nor reaches cache full conditions for the same workload.

amount of time. Since there are no large fluctuations in STOW’s cache occupancy, STOW delivers a consistent performance with any threshold, beating the best configuration for either WOW or CSCAN.

In particular, at around 20ms response time, with a threshold of 90/80, in terms of SPC-1 Like IOPS in RAID-5, STOW outperforms WOW by 70%, CSCAN by 96%, and LRW by 39%. With a threshold of 70/40, STOW beats WOW by 18%, CSCAN by 26%, and LRW by 39%. Similarly, in RAID-10 with a 90/80 threshold, STOW outperforms WOW by 40%, CSCAN by 53%, and LRW by 27%, while, with a threshold of 70/40, STOW beats WOW by 20%, and CSCAN and LRW by 27%. These gains are not trivial in the world of hard drives which sees only a meager improvement rate of 8% per year. Although we include data points at response times greater than 30ms, they are not of much practical significance as applications would become very slow at those speeds. Even the SPC-1 Benchmark disallows submissions with greater than 30ms response times.

6.2.2 Partial Backend

For the partial backend scenario, depicted in the lower panel in Figure 8, we use only the outer 1/100th of each disk in the RAID array, creating a high hit ratio scenario with short-stroking on the disks. In this setup, the fluctuations in the occupancy for WOW are closer together (Figure 7(b)), resulting in a more rapid alternation between sequential and random destages. This helps WOW somewhat, however, in terms of SPC-1 Like IOPS at 20ms, STOW still beats WOW by 12%, CSCAN by 160%, and LRW by 24% in a RAID-5 setup. In the RAID-10 setup, where writes become less important (because of no read-modify-write penalties), STOW still

beats WOW by 3% (actually it is much better at lower response times), CSCAN by 120%, and LRW by 42%.

6.2.3 WOW’s thresholding dilemma

	Full Backend		Partial Backend	
	H/L: 90/80	70/40	H/L: 90/80	70/40
STOW	5.18	5.58	1.28	2.23
WOW	22.69	5.78	1.38	2.26
CSCAN	27.19	6.03	41.32	24.08
LRW	6.14	6.58	1.50	2.23

Table 2: Response times (in milliseconds) at lower throughputs (better numbers in bold). For full backend, the response times correspond to a target of 750 IOPS from Figure 8(a), and for partial backend to a target of 2000 IOPS from Figure 8(c). WOW’s best threshold choice, unlike STOW, depends on the backend setup.

The response time in a lightly loaded system is also an important metric [14]. We present the actual numbers corresponding to RAID-5 in Figure 8 in Table 2.

Note that in all cases, STOW beats the competition easily. However, WOW is unique in that it requires different thresholds to perform its best for different backend scenarios. While, choosing a conservative (70/40) threshold allows WOW to beat CSCAN and LRW, WOW is forced to sustain a response time of 2.26 ms in the partial backend case, even though it could have delivered 1.38 ms response time with a higher threshold which allows for higher hit ratio. Since the workload is not known *a priori*, the right choice of threshold levels remains elusive for WOW. So, in real life, STOW would cut the response time not by just 7% (1.28 ms vs 1.38 ms) when compared to WOW, but rather by 43% (1.28 ms vs

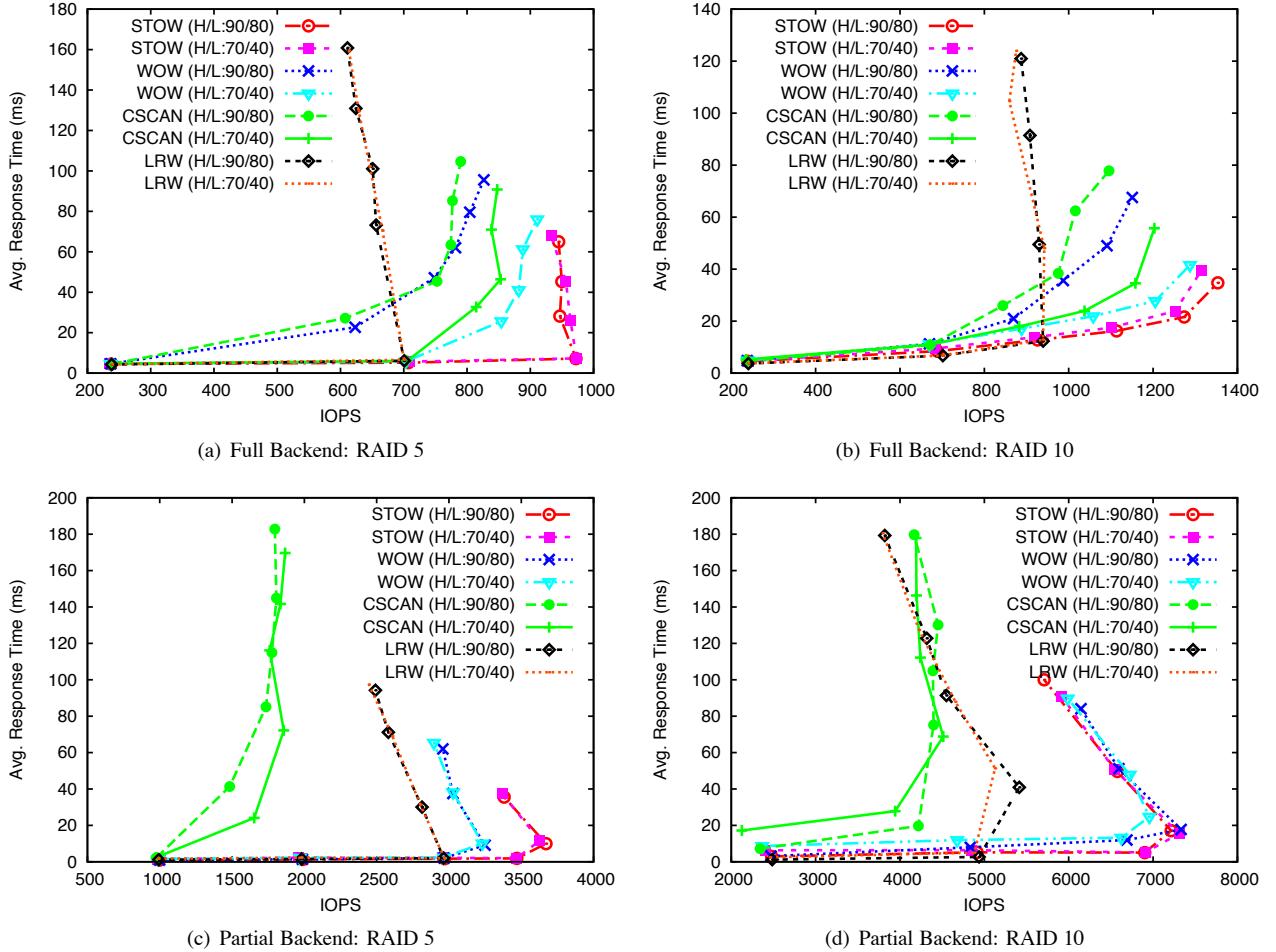


Figure 8: We increase the target throughput for the SPC1-Like Benchmark and present the achieved throughput as a function of the aggregate (read and write) response time for both the 90/80 and 70/40 destage thresholds in a 32K page cache. Each data point is the average of measurements over 5 minutes after 5 mintues of warmup time. While WOW beats LRW and CSCAN, STOW outperforms WOW consistently.

2.26 ms). An adaptive threshold determination scheme might help WOW somewhat, but in no instance would it be able to compete with STOW, which at the fixed 90/80 threshold consistently outperforms its competition.

6.3 Varying Threshold Level

In Figure 9, we examine how changing the thresholds alone while keeping the workload constant changes the performance of a write cache. For WOW and CSCAN, in the full backend case, we can clearly see that as the thresholds become lower, the performance improves. While the lower thresholds help keep the occupancy fluctuations away from 100% occupancy more effectively, it cannot completely eradicate the phenomenon and, consequently, both WOW and CSCAN fare worse than STOW. STOW beats WOW and CSCAN by 19% on average, and

LRW by 46% in terms of SPC-1 Like IOPS.

In the partial backend case, both WOW and LRW are better than CSCAN because they can leverage temporal locality more effectively. Further, the performance does not depend on the choice of the threshold in this case because what is gained by keeping lower thresholds is lost in the extra misses incurred in this high hit ratio scenario. In terms of SPC-1 Like IOPS, STOW beats WOW by about 7%, LRW by 22%, and CSCAN by about 96%.

6.4 Varying Cache Size

Any good adaptive caching algorithm should be able to perform well for all cache sizes. In Figure 10(a), we observe that for the full backend scenario, across all cache sizes, STOW outperforms WOW by 17-30%, CSCAN by 19-40%, and LRW by 35-48%. The gains are more

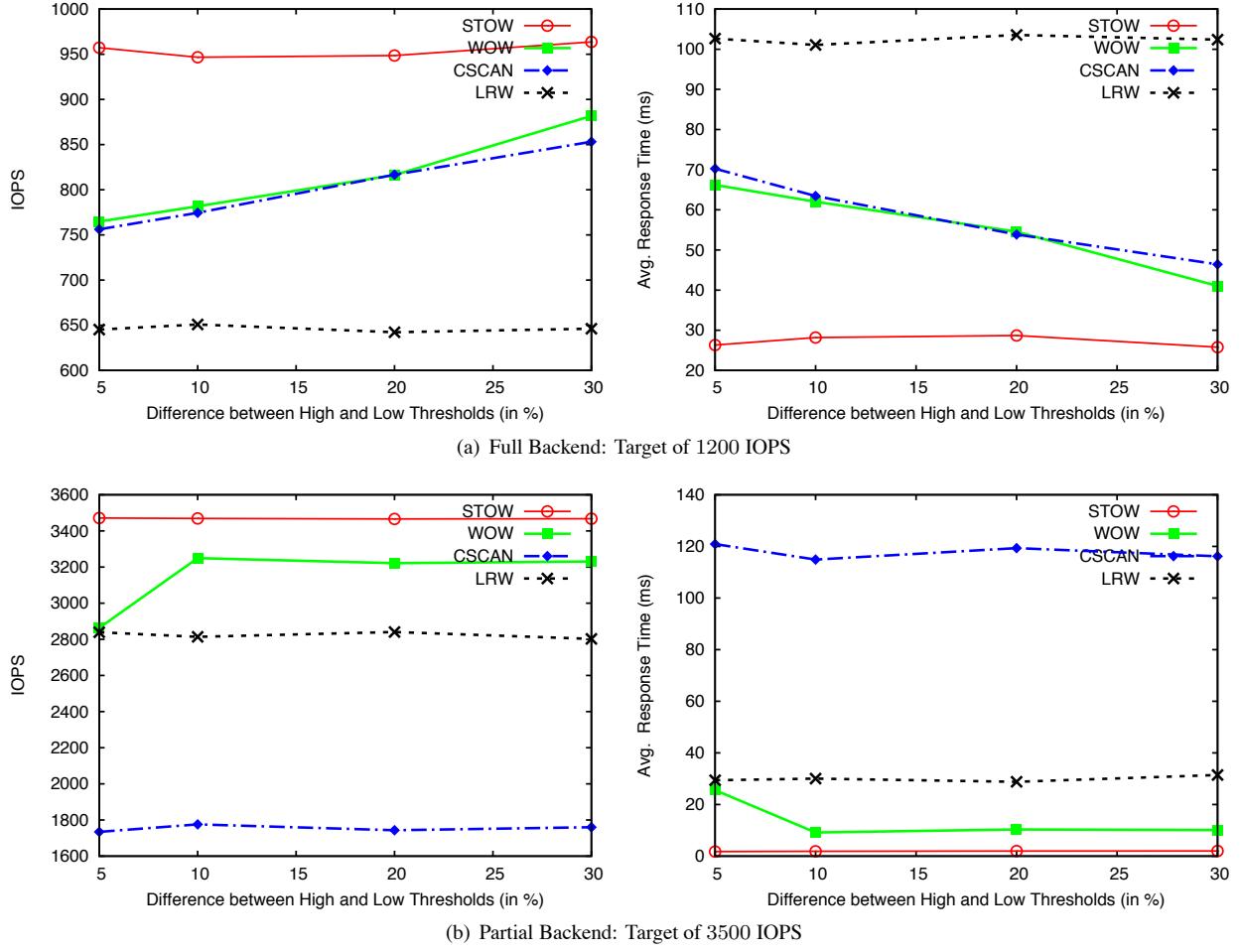


Figure 9: We vary the spread between the high and low thresholds while keeping target workload fixed (32K page cache, RAID-5). The left panel shows the measured throughput and the right panel the corresponding average response times. RAID-10 has similar results.

significant for larger caches because of two reasons: (i) a larger cache causes the cache occupancy spikes in WOW and CSCAN to be further apart and much larger in amplitude, making it easier to hit the full-cache condition (the performance of CSCAN actually dips as the cache size increases to 131072 pages!); (ii) a larger cache in LRW, WOW, and CSCAN proportionally devotes more cache space to sequential data even though there might be nothing to gain. STOW adapts the sizes of SeqQ and RanQ, which limits the size of SeqQ in larger caches, and creates better spatial locality in the larger RanQ.

The partial backend scenario, presented in Figure 10(b), also indicates that STOW is the best algorithm overall. With smaller cache sizes, the lower hit ratio overdrives the cache for all algorithms resulting in very high response times, which are not of much practical interest. If we had scaled the workload according to what the cache could support, the benefit of STOW

would be seen consistently even for lower cache sizes. At a cache size of 32K pages, in terms of SPC-1 Like IOPS, STOW outperforms WOW by 21%, CSCAN by 104%, and LRW by 43%. The performance at higher cache sizes is similar for all algorithms because the working set fits in the cache, eliminating the disk bottleneck.

7 Conclusions

STOW represents a significant improvement over the state of the art in write caching algorithms. While write caching algorithms have mainly focused on the order of destages, we have shown that it is critical to wisely control the rate of destages as well. STOW outperforms WOW by a wider margin than WOW outperforms CSCAN and LRW. The observation that the order of destages needs to change to accommodate a better control on the rate of destages is a key one. We hope that we

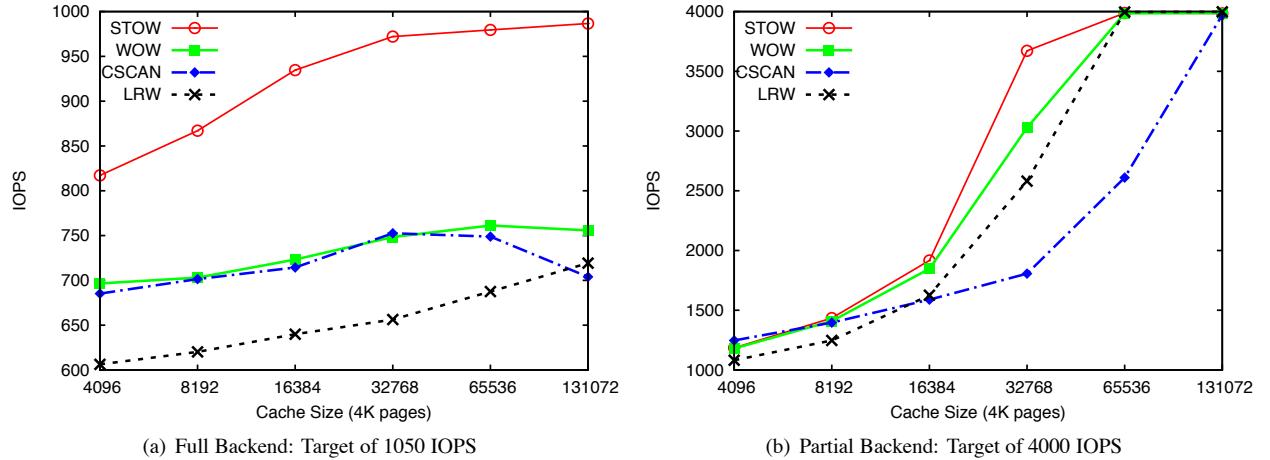


Figure 10: Measured throughput as we vary cache size in a RAID-5 setup (RAID-10 is similar) with 90/80 thresholds.

have furthered the appreciation of the multi-dimensional nature of the write caching problem, which will spark new efforts towards advancements in this critical field.

References

- [1] BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SELTZER, M. Non-volatile memory for fast, reliable file systems. *SIGPLAN Not.* 27, 9 (1992), 10–22.
- [2] BANSAL, S., AND MODHA, D. S. CAR: Clock with Adaptive Replacement. In *Proc. Third USENIX Conf. on File and Storage Technologies (FAST 04)* (2004), pp. 187–200.
- [3] BISWAS, P., RAMAKRISHNAN, K. K., TOWSLEY, D. F., AND KRISHNA, C. M. Performance analysis of distributed file systems with non-volatile caches. In *Proc. 2nd Int'l Symp. High Perf. Distributed Computing* (1993), pp. 252–262.
- [4] COFFMAN, E. G., KLIMKO, L. A., AND RYAN, B. Analysis of scanning policies for reducing disk seek times. *SIAM J. Comput.* 1, 3 (1972), 269–279.
- [5] CORBATO, F. J. A paging experiment with the Multics system. Tech. rep., Massachusetts Inst. of Tech. Cambridge Project MAC, 1968.
- [6] DENNING, P. J. Effects of scheduling on file memory operations. In *Proc. AFIPS Spring Joint Comput. Conf.* (1967), pp. 9–21.
- [7] GEIST, R., AND DANIEL, S. A continuum of disk scheduling algorithms. *ACM Trans. Comput. Syst.* 5, 1 (1987), 77–92.
- [8] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in Adaptive Replacement Cache. In *Proc. USENIX 2005 Annual Technical Conf. (USENIX)* (2005), pp. 293–308.
- [9] GILL, B. S., AND MODHA, D. S. WOW: Wise Ordering for Writes - combining spatial and temporal locality in non-volatile caches. In *Proc. Fourth USENIX Conf. on File and Storage Technologies (FAST 05)* (2005), pp. 129–142.
- [10] HAINING, T. R. *Non-volatile cache management for improving write response time with rotating magnetic media*. PhD thesis, University of California, Santa Cruz, 2000.
- [11] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. I/O reference behavior of production database workloads and the TPC benchmarks—an analysis at the logical level. *ACM Trans. Database Syst.* 26, 1 (2001), 96–143.
- [12] JACOBSON, D., AND WILKES, J. Disk scheduling algorithms based on rotational position. Tech. Rep. HPL-CSP-91-7rev1, HP Labs, February 1991.
- [13] JIANG, S., AND ZHANG, X. LIRS: An efficient Low Interference Recency Set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and modeling of computer systems* (2002), pp. 31–42.
- [14] JOHNSON, S., MCNUTT, B., AND REICH, R. The making of a standard benchmark for open system storage. *J. Comput. Resource Management*, 101 (2001), 26–32.
- [15] LEE, D., CHOI, J., KIM, J.-H., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and modeling of computer systems* (1999), pp. 134–143.
- [16] MCNUTT, B., AND JOHNSON, S. A standard test of I/O cache. In *Proc. Int'l CMG Conference* (2001), pp. 327–332.
- [17] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proc. Second USENIX Conf. on File and Storage Technologies (FAST 03)* (2003), pp. 115–130.
- [18] MERTEN, A. G. *Some quantitative techniques for file organization*. PhD thesis, Univ. of Wisconsin, June 1970.
- [19] ROBINSON, J. T., AND DEVARAKONDA, M. V. Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and modeling of computer systems* (1990), pp. 134–142.
- [20] SEAMAN, P. H., LIND, R. A., AND WILSON, T. L. On teleprocessing system design Part IV: An analysis of auxiliary storage activity. *IBM Systems Journal* 5, 3 (1966), 158–170.
- [21] SELTZER, M., CHEN, P., AND OUSTERHOUT, J. Disk scheduling revisited. In *Proc. USENIX Winter 1990 Tech. Conf.* (1990), pp. 313–324.
- [22] VARMA, A., AND JACOBSON, Q. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Trans. Comput.* 47, 2 (1998), 228–235.
- [23] VARMA, A., AND JACOBSON, Q. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Trans. Computers* 47, 2 (1998), 228–235.
- [24] ZHOU, Y., CHEN, Z., AND LI, K. Second-level buffer cache management. *IEEE Trans. Parallel and Distrib. Syst.* 15, 6 (2004), 505–519.

Black-Box Performance Control for High-Volume Non-Interactive Systems

Chunqiang Tang¹, Sunjit Tara², Rong N. Chang¹, and Chun Zhang¹

¹ IBM T.J. Watson Research Center ² IBM Software Group, Tivoli
`{ctang, sunjit.tara, rong, czhang1}@us.ibm.com`

Abstract

This paper studies performance control for high-volume non-interactive systems, and uses IBM Tivoli Netcool/Impact—a software product in the IT monitoring and management domain—as a concrete example. High-volume non-interactive systems include a large class of applications where requests or processing tasks are generated automatically in high volume by software tools rather than by interactive users, e.g., data stream processing and search engine index update. These systems are becoming increasingly popular and their performance characteristics are radically different from those of typical online Web applications. Most notably, Web applications are response time sensitive, whereas these systems are throughput centric.

This paper presents a performance controller, TCC, for throughput-centric systems. It takes a black-box approach to probe the achievable maximum throughput that does not saturate any bottleneck resource in a distributed system. Experiments show that TCC performs robustly under different system topologies, handles different types of bottleneck resources (e.g., CPU, memory, disk, and network), and is reactive to resource contentions caused by an uncontrolled external program.

1 Introduction

Performance control for online interactive Web applications has been a research topic for years, and tremendous progress has been made in that area [2, 10, 23, 28]. By contrast, relatively little attention has been paid to performance control for a large class of increasingly popular applications, where requests or processing tasks are generated automatically in high volume by software tools rather than by interactive users. Many emerging stream processing systems [1] fall into this category, e.g., continuous analysis and distribution of news articles, as in Google Reader [11] and System S [19].

Moreover, almost every high-volume interactive Web application is supported behind the scene by a set of high-volume non-interactive processes, e.g., Web crawling and index update in search engines [7], Web log mining for portal personalization [22], video preprocessing and format conversion in YouTube, and batch conversion of rich-media Web sites for mobile phone users [3].

Beyond the Web domain, examples of high-volume non-interactive systems include IT monitoring and management [15], overnight analysis of retail transaction logs [5], film animation rendering [14], scientific applications [6], sensor networks for habitat monitoring [20], network traffic analysis [26], and video surveillance [8].

The workloads and operating environments of these high-volume non-interactive systems differ radically from those of session-based online Web applications. Most notably, Web applications usually use response time to guide performance control [2, 23, 28], whereas high-volume non-interactive systems are throughput centric and need not guarantee sub-second response time, because there are no interactive users waiting for immediate responses of *individual* requests. Instead, these systems benefit more from high throughput, which helps lower *average* response time and hardware requirements.

This paper studies performance control for high-volume non-interactive systems, and uses IBM Tivoli Netcool/Impact [16]—a software product in the IT monitoring and management domain—as a concrete example.

Today's enterprise IT environments are extremely complex. They often include resources from multiple vendors and platforms. Every hardware, OS, middleware, and application usually comes with its own siloed monitoring and management tool. To provide a holistic view of the entire IT environment while taking into account dependencies between IT components, a federated IT Service Management (ITSM) system may use a core event-processing engine such as Netcool/Impact to drive and integrate various siloed software involved in IT management.

An IT event broadly represents a piece of information that need be processed by the ITSM system. For instance, under normal operations, transaction response time may be collected continuously to determine the service quality. Monitoring tools can also generate events to report problems, e.g., a database is down. When processing an event, the event-processing engine may interact with various third-party programs, e.g., retrieving customer profile from a remote database and invoking an instant messaging server to notify the system administrator if a VIP customer is affected.

When a major IT component (e.g., core router) fails, the rate of IT events may surge by several orders of mag-

nitude due to the domino effect of the failure. If the event-processing engine tries to process all events concurrently, either the engine itself or some third-party programs working with the engine may become severely overloaded and suffer from thrashing. In this work, the purpose of performance control is to dynamically adjust the concurrency level in the event-processing engine so as to maximize throughput while avoiding fully saturating either the engine itself or any third-party program working with the engine, i.e., targeting 85–95% resource utilization (as opposed to 100%) even during peak usage.

The main difficulty in achieving this goal is caused by the diversity and proprietary nature of the multi-vendor components used in a federated ITSM system. For practical reasons, we can only take a black-box approach and cannot rely on many assumptions presumed by existing performance control algorithms.

- We cannot aggressively maximize performance without considering resource contention with external programs not under our control. Therefore, we cannot use greedy parameter search [25].
- We cannot assume a priori knowledge of system topology (e.g., three-tier), and hence cannot use solutions based on static queueing models [23].
- We cannot assume knowledge of every external program’s service-level objectives (as in [18]), or knowledge of every component’s performance characteristics, e.g., through offline profiling as in [24]. Therefore, we cannot directly adopt these methods based on classical control theory.
- We cannot assume the ability to track resource consumption of every component, or a prior knowledge of the location or type of the bottleneck. Therefore, we cannot adopt solutions that adjust program behavior based on measured resource utilization level.
- We have no simple performance indicators to guide tuning, such as packet loss in TCP [17] or response time violation in interactive Web applications [2].

1.1 Throughput-guided Concurrency Control

The discussion below assumes that the number of worker threads in the event-processing engine controls the concurrency level. We explore the relationship between throughput and the event-processing concurrency level to guide performance tuning (see Figure 1). With too few threads, the throughput is low while system resources are underutilized. As the number of threads increases, the throughput initially increases almost linearly, and then gradually flattens, because the bottleneck resource is near saturation. Once the bottleneck saturates, adding more threads actually decreases throughput because of the overhead in managing resource con-

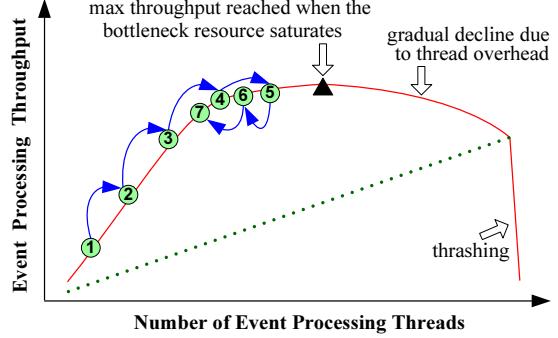


Figure 1: Basic idea of throughput-guided concurrency control (TCC). The symbols ①–⑦ show the controller’s operation sequence. If memory is the bottleneck resource, the throughput may follow the dotted line and then suddenly move into thrashing without a gradual transition. This figure is further explained in Section 3.1.

tension. Finally, using an excessive number of threads causes thrashing, and the throughput drops sharply.

We refer to our controller as **TCC** (throughput-guided concurrency control). Intuitively, it works as follows. Starting from an initial configuration, it tentatively adds some threads (transition ①→② in Figure 1), and then compares the throughput measured before and after the change. If the throughput increases “significantly”, it keeps adding threads (transitions ②→③→④), until either the throughput starts to decline or the improvement in throughput becomes marginal (transition ④→⑤). It then successively removes threads (transitions ⑤→⑥→⑦), until the throughput becomes a certain fraction (e.g., 95%) of the maximum throughput achieved during the exploration. The purpose is to reach a stable state that delivers high throughput while not saturating the bottleneck resource.

We address several challenges to make this basic idea practical. Because the exact shape of the thread-throughput curve in Figure 1 varies in different environments, a robust method is needed to determine when the throughput “almost” flattens. If the controller adds threads too aggressively, it may cause resource saturation and gain unfair advantages when competing with an uncontrolled external program. Another challenge is to make quick control decisions based on noisy performance measurement data, e.g., abnormal long pauses caused by Java garbage collection. Our solutions to these challenges are described in Section 3.

Our controller is flexible. It takes a black-box approach to maximize throughput while trying to avoid saturating the bottleneck resource. It makes few assumptions about the operating environment. It need not know system topology, performance characteristics of external programs, resource utilization level, or exactly which resource is the bottleneck. It can handle both hardware (e.g., CPU, memory, disk, or network) and software (e.g.,

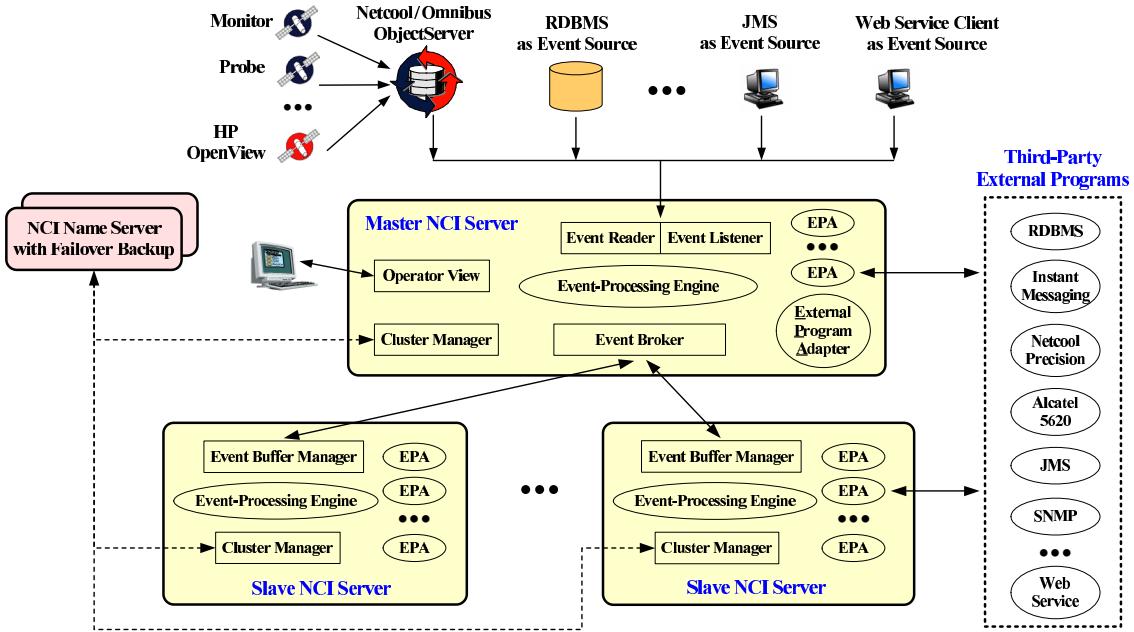


Figure 2: A radically simplified architecture diagram of Netcool/Impact (NCI). IT events flow from top to bottom.

database connection pool) bottleneck resources. Because of its flexibility, it may be broadly applied to high-volume non-interactive systems [1, 6, 7, 19, 20, 22, 26].

We have implemented TCC and integrated it with IBM Tivoli Netcool/Impact [16]. The Netcool suite [15] is a set of software products that help implement a federated ITSM system, and Netcool/Impact is the streaming event-processing engine of the Netcool suite. Experiments demonstrate that TCC performs robustly under different system topologies, handles different types of bottleneck resources, and is reactive to resource contentions caused by an uncontrolled external program.

The remainder of the paper is organized as follows. Section 2 provides an overview of Netcool/Impact. Sections 3 and 4 present and evaluate TCC, respectively. Related work is discussed in Section 5. Section 6 concludes the paper.

2 Overview of Netcool/Impact (NCI)

Our performance controller is generic, and its current implementation can actually be configured to compile and run independent of the Netcool/Impact product. To make the discussion more concrete, however, we choose to present it in the context of Netcool/Impact. Netcool/Impact is a mature product with a large set of features. Below, we briefly summarize those features most relevant to our discussion. We simply refer to Netcool/Impact as NCI.

NCI adopts a clustering architecture (see Figure 2). The “master NCI server” is the data fetcher and load balancer. Its “event reader” pulls IT events from various sources, while its “event listener” receives events pushed from various sources. It processes some events in its lo-

cal “event-processing engine”, and dispatches the rest to the “slave NCI servers” for load balancing. The “NCI name server” manages members of the cluster. If the master fails, a slave will be converted into master.

The “event-processing engine” executes user-supplied programs written in the Impact Policy Language (IPL). IPL is a proprietary scripting language specially designed for event processing, emphasizing ease of use for system administrators. With the help of various built-in “external program adapters”, IPL scripts can easily integrate with many third-party programs.

Each NCI server (master or slave) uses a pool of threads to process events and runs a performance controller to determine for itself the appropriate thread pool size. When an event arrives, the NCI server goes through a list of admin-specified matching rules to identify the IPL script that will be used to process the event. The event waits in a queue until an event-processing thread becomes available, and then the thread is dispatched to interpret the IPL script with the event as input.

In a large IT environment, monitoring events (e.g., CPU utilization reports) are generated continuously at a high rate even under normal operations. Some events are filtered locally, while the rest are collected in realtime, e.g., to the Netcool/OMNIBUS ObjectServer [15], which buffers events and feeds them to the master NCI server in batches, e.g., one batch every five seconds. Events are not sent to the master individually for the sake of efficiency. Similarly, a slave NCI server fetches events in batches from the master.

Because events are fetched in batches, an NCI server often holds a large number of unprocessed events. If the server tries to process all of them concurrently, either

the server itself or some third-party programs working with the server will become severely overloaded and suffer from thrashing. Moreover, it needs to carefully control the concurrency level of event processing so that it achieves high throughput while sharing resources with a competing program in a friendly manner.

Our performance control goal is to maximize event-processing throughput while avoiding saturating NCI or any third-party program working with NCI. Even during peak usage, the utilization level of the bottleneck resource should be controlled, e.g., between 85% and 95%, instead of 100%. We must avoid saturating the master NCI server because it hosts other services such as “operator view” (see Figure 2), which provides a customizable dashboard for administrators to look into the details of IT events. In addition, we must avoid saturating third-party programs working with NCI, because they may serve clients other than NCI, including interactive users.

In light of today’s complex and heterogeneous IT environments, the success of the NCI product to a great extent owes to its common adapter platform that helps integrate various distributed data sources and siloed monitoring and management tools. Because of the diversity and proprietary nature of these third-party external programs working with NCI, we can only take a black-box approach and cannot rely on many assumptions that are presumed by existing performance control algorithms (as those listed in Section 1).

3 Our Performance Control Algorithm

This section presents our controller TCC in detail. We start with a description of TCC’s state transition diagram, and then use queuing models to analyze TCC and demonstrate that it can achieve high resource utilization. We then derive the friendly resource sharing conditions for TCC. We also present a statistical method that minimizes measurement samples needed for making control decisions. Finally, we put all the pieces together to guide the selection of TCC’s parameters.

3.1 State Transition Diagram

TCC operates according to the state-transition diagram in Figure 3. Most of the time, it stays in the “steady” state, using a constant number of threads to process events that continuously arrive in batches. The number of threads is optimal if those threads can drive the bottleneck resource to a high utilization level (e.g., 85-95%) while avoiding fully saturating it.

Periodically, TCC gets out of the steady state to explore whether a better configuration exists. It moves into the “base” state and reduces the number of threads by $w\%$, which will serve as the exploration starting point ① in Figure 1. (How to select parameters such as $w\%$ will be discussed in Section 3.5.) TCC stays in the “base” state for a short period of time to measure the event-processing throughput. It then increases the number of

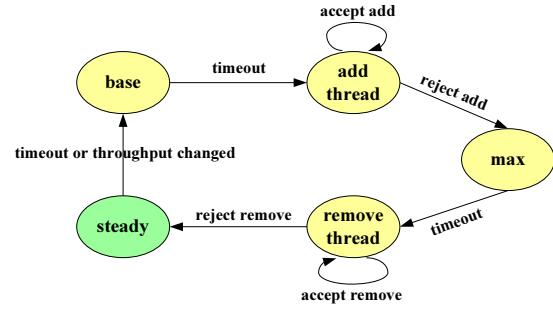


Figure 3: Simplified state-transition diagram of TCC.

threads by $p\%$ and moves into the “add-thread” state. If this $p\%$ increase in threads helps improve throughput by $q\%$ or more, it stays in the add-thread state and repeatedly add threads by $p\%$ each time. Eventually, the bottleneck resource is near saturation so that a $p\%$ increase in threads no longer gives a $q\%$ or more increase in throughput. It then moves into the “max” state.

TCC takes more measurement samples in the “max” state in order to calculate a more accurate baseline throughput. It then moves into the “remove-thread” state to repeatedly removes threads by $r\%$ each time so long as the throughput does not drop below 95% of the highest throughput achieved during the current tuning cycle.

When the throughput finally drops below the 95% threshold, it adds back threads removed in the last round, and moves into the steady state. It stays in the steady state for a relatively long period of time, using an optimal number of threads to process events. It restarts the next round of exploration either after a timeout or when the throughput changes significantly, which indicates a change in the operating environment.

If memory is the bottleneck, throughput may follow the dotted line in Figure 1, and then suddenly moves into thrashing when TCC adds threads. TCC will detect the decline in throughput, revoke the threads just added, and continue to remove more threads until the throughput becomes 95% of the measured maximum throughput. This prevents the system from moving into thrashing.

3.2 High Resource Utilization

In this section, we use queuing models to demonstrate that, for common event processing scenarios, TCC can achieve high resource utilization (and hence high throughput) while avoiding resource saturation. The discussion below assumes that TCC uses the default configuration: $p=25\%$, $q=14\%$, and $w=39\%$. (Section 3.5 discusses parameter selection.) Our queuing models assume the ITSM system consists of one NCI server and some third-party external programs. We are interested in system behavior when it continuously processes a block of events and we assume no threads remain idle due to the lack of input events.

The first model we use is the machine-repairman model [12] in Figure 4(a). This model assumes that the

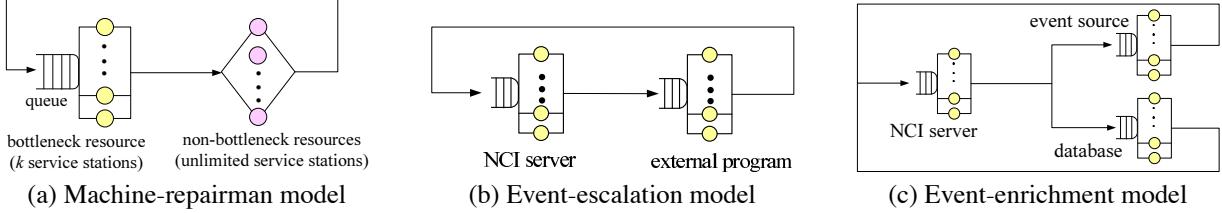


Figure 4: Using queueing models to analyze TCC. These closed models can be solved by mean value analysis [12].

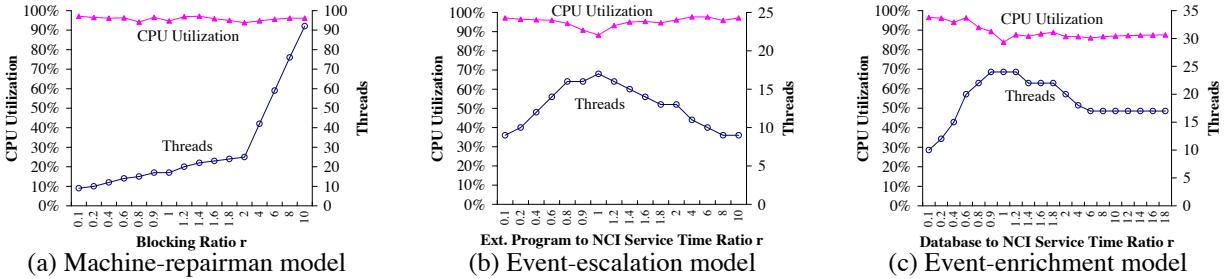


Figure 5: Performance of TCC under different queueing models. Note that the x -axis increases nonlinearly.

ITSM system has a clearly defined bottleneck resource, whose utilization level is much higher than that of the other resources. Even if the bottleneck is fully saturated, the other resources are still underutilized. Therefore, the queueing delays of the non-bottleneck resources can be approximately ignored. We use machine-repairman model’s delay station to represent the sum of the service times of all non-bottleneck resources. As the delay station can abstractly represent multiple distributed resources, real systems of different topologies (e.g., 3 machines or 7 machines) can be represented by this single model, so long as they have a clearly defined bottleneck. Many real systems do satisfy this requirement.

The machine-repairman model can predict event-processing throughput and resource utilization level under different thread configurations [12]. We modified our implementation of TCC to take throughput numbers from the model instead of a live system. This allows us to systematically evaluate TCC under a wide range of hypothetical workloads.

Figure 5(a) shows the number of threads recommended by TCC and the corresponding CPU utilization level. Here we assume that the CPU of the NCI server is the bottleneck resource, and it has 8 CPUs. CPU utilization is affected by the blocking ratio r , which is defined as the service time ratio of the delay station to the bottleneck resource. As r increases, each thread blocks longer at the delay station, and hence more threads are needed to drive up CPU utilization. As r varies, TCC is able to adjust the number of threads accordingly to keep high CPU utilization while avoiding complete saturation.

Figure 4(b) shows the event-escalation model, where the NCI server processes an event and then invokes an external program, e.g., an instant messaging server.

This model differs from the machine-repairman model in that it does not assume the queueing delays of the non-bottleneck resources are negligible. Figure 5(b) shows the performance of TCC when both machines have 8 CPUs. The x -axis is the service time ratio r of the external program to the NCI server. The y -axis is the CPU utilization of the bottleneck resource. The bottleneck is the NCI server if $r < 1$, or the external program if $r > 1$. The lowest utilization 88% occurs when $r = 1$, i.e., when the utilization levels of two machines are identical. In this case, more threads are needed to simultaneously drive both machines to high utilization.

Figure 4(c) shows the event-enrichment model, where the NCI server processes an event, enriches it with data fetched from an external database, and writes it back to the event source. This is a widely used topology in real deployments. Figure 5(c) shows the performance of TCC when each of the three machines has have 8 CPUs. The x -axis is the service time ratio r of the database to the NCI server. The database and the event source have the same mean service time. The y -axis is the utilization of the bottleneck. The lowest utilization 85% occurs when $r = 1$, i.e., when the utilization levels of the three machines are identical.

Results in Figure 5 show that TCC can drive the bottleneck resource to high utilization under different workloads and deployment topologies. On the other hand, TCC may underutilize resources in some cases, e.g., when processing one event goes through a large number of servers whose utilization levels are identical (i.e., $r=1$). To reduce resource waste in this worst case, one might be tempted to make TCC more aggressive in adding threads. However, this would also make TCC less friendly in resource sharing.

3.3 Friendly Resource Sharing

Below, we derive the conditions for friendly resource sharing and demonstrate that, with a proper configuration, TCC shares resources in a friendly manner with an uncontrolled competing program. Moreover, multiple instances of TCC also share resources in a friendly manner. We begin our discussion with the basic two-NCI-server scenario.

Suppose two NCI servers independently execute TCC. If each server has its own internal bottleneck that limits its throughput, TCC will independently drive each server to almost full utilization. A more challenging case is that a shared bottleneck resource limits the throughput of both NCI servers, e.g., a shared database. Below, we will show that, when the shared bottleneck is saturated, the two NCI servers take turns to reduce their threads until the bottleneck resource is relieved of saturation.

Suppose the bottleneck resource is fully saturated, two NCI servers X and Y are identical, and they currently run x_0 and y_0 threads, respectively, where $x_0 \leq y_0$. A TCC tuning cycle consists of the tuning steps starting from the base state and finally settling in the steady state. We use i to number TCC's tuning cycles in increasing order, and assume X and Y take turns to execute in the tuning cycles, i.e., if X executes in cycle i , then Y will execute in cycle $i + 1$. Let x_i and y_i denote the numbers of X and Y 's threads at the end of tuning cycle i .

Theorem 1 *If TCC's parameters p , q , and w satisfy Equations (1) and (2) below, then X and Y will take turns to reduce their threads (i.e., $y_0 > x_1 > y_2 > x_3 \dots$) until the bottleneck is relieved of saturation.*

$$q > \frac{p(p+1)}{p+2} \quad (1)$$

$$w \geq 1 - \left(\frac{p}{q} - 1\right)^2 \quad (2)$$

Moreover, if TCC's parameters satisfy (1) and (2), a TCC instance shares resources with an external competing program in a friendly manner.

Proof sketch: Suppose X is in the process of tuning its configuration, and just finished increasing its threads from $\frac{x}{1+p}$ to x . When X uses x threads to compete with Y 's y_0 threads, X 's throughput is $f(x, y_0) = \frac{x}{x+y_0}C$, where C is the maximum throughput of the bottleneck. TCC keeps adding threads so long as every $p\%$ increase in threads improves throughput by $q\%$ or more. Therefore, X continues to add more threads if and only if

$$\frac{f(x, y_0)}{f(\frac{x}{1+p}, y_0)} \geq 1 + q, \quad (3)$$

which is equivalent to $x \leq (\frac{p}{q} - 1)y_0$. Let \hat{y} denote the upper bound of this condition:

$$\hat{y} = \left(\frac{p}{q} - 1\right)y_0. \quad (4)$$

Suppose X runs no more than \hat{y} threads in the base state. (This assumption holds if (2) holds.) X keeps adding threads so long as its current number of threads is no more than \hat{y} . Hence, when X stops adding threads, its final number x_1 of threads falls into the range

$$\hat{y} < x_1 \leq (1 + p)\hat{y}. \quad (5)$$

X ends up with fewer threads than Y if $(1 + p)\hat{y} < y_0$. From (4), this condition is equivalent to (1).

When X uses x_1 threads to compete with Y 's y_0 threads, X 's share of the bottleneck is bounded by

$$1 - \frac{q}{p} < \frac{x_1}{x_1 + y_0} \leq \frac{(1 + p)(p - q)}{p(1 + p - q)}. \quad (6)$$

This bound is derived from (4) and (5).

Now suppose Y executes TCC after X settles with x_1 threads. Y first reduces its threads by $w\%$ in the base state. Following (4), we define

$$\hat{x} = \left(\frac{p}{q} - 1\right)x_1. \quad (7)$$

If Y 's base state has no more than \hat{x} threads, i.e., if

$$(1 - w)y_0 \leq \hat{x}, \quad (8)$$

then we can follow (5) to obtain the bound of Y 's final number y_2 of threads when Y stops adding threads:

$$\hat{x} < y_2 \leq (1 + p)\hat{x}. \quad (9)$$

From (4), (5), and (7), we know that (8) holds if (2) holds.

TCC's default parameters are $p=25\%$, $q=14\%$, and $w=39\%$, which satisfy (1) and (2). Therefore, it follows from (5) and (9) that $y_0 > x_1 > y_2$. This reduction in threads continues as X and Y repeatedly execute TCC, until the bottleneck is relieved of saturation.

Following the approach above, one can also show that TCC shares resources in a friendly manner with an external competing program that generates a constant workload at the shared bottleneck resource. In the face of competition, TCC dynamically adjusts the number of processing threads so that it consumes about 44–49% of the bottleneck resource. This range is obtained by substituting the default parameters ($p=25\%$ and $q=14\%$) into (6). By contrast, if one uses a configuration that does not satisfy (1), TCC's consumption of the bottleneck resource could be unfairly high, e.g., reaching 80–83% for the configuration $p=25\%$ and $q=5\%$.

The analysis above focuses on the base state and the add-thread state. The remove-thread state removes threads to avoid saturation, which makes TCC even more friendly in resource sharing. Therefore, Theorem 1 holds if the remove-thread state is taken into account. ■

With a proper configuration, a TCC instance shares resources in a friendly manner with an external competing program, and two TCC instances also share resources in a friendly manner. Three or more instances of TCC share resources in a friendly manner only if they execute

in a loosely synchronized fashion, i.e., they move out of the steady state into the base state roughly at the same time. When the shared bottleneck is saturated and multiple TCC instances attempt to add threads at the same time, they will observe little improvement in throughput and gradually remove threads until the bottleneck is relieved of saturation. A detailed analysis is omitted here. In an NCI cluster, the master can serve as the coordinator to enforce loose synchronization. Using loosely synchronized execution to enforce friendly resource sharing has also been proposed in Tri-S [27], although its application domain is TCP congestion control.

3.4 Accurate Performance Measurement

TCC repeatedly adds threads so long as every $p\%$ increase in threads improves throughput by $q\%$ or more. Let C_1 and C_2 denote the configurations before and after adding the $p\%$ threads, respectively. (This section uses the add-thread state as example. The remove-thread state can be analyzed similarly.) In a noisy environment, throughput is a stochastic process and accurate measurement is challenging. On the one hand, the throughput of a configuration can be measured more accurately if TCC stays in that configuration longer and takes more measurement samples. On the other hand, we want to minimize the measurement time so that TCC responds to workload changes quickly.

We formulate the issue of accurate performance measurement as an optimization problem. The optimization goal is to minimize the total number of samples collected from configurations C_1 and C_2 , and the constraint is to ensure a high probability of making a correct control decision. It turns out that the number of samples needed to make a reliable decision is proportional to the variance of event-processing time (i.e., more samples are needed if the system is volatile), and inversely proportional to the throughput improvement threshold q (i.e., more samples are needed if we want to tell even a small performance difference between two configurations).

Below, we present our statistical approach for performance measurement, our method for handling unstable event arrival rate, and our heuristic for filtering out large noises caused by extreme activities such as Java garbage collection.

3.4.1 Our Experiment Design Approach

We use subscript i to differentiate the two configurations C_i , $i = 1, 2$. For configuration C_i , let random variable X_i denote the inter-departure time between the completion of event processing. Denote μ_i and σ_i^2 the mean and variance of X_i . Suppose we take n_i samples of X_i , denoted as X_{ij} , $1 \leq j \leq n_i$, and these samples are independent and identically distributed. Denote \bar{X}_i the sample mean of X_{ij} . According to the central limit theorem, regardless of the distribution of X_i , \bar{X}_i is approximately normally distributed, $\bar{X}_i \sim N(\mu_i, \sigma_i^2/n_i)$.

Let $Y = \bar{X}_1 - \bar{X}_2$, which represents the performance difference between C_1 and C_2 . Assuming \bar{X}_1 and \bar{X}_2 are independent, Y is also approximately normally distributed, $Y \sim N(\mu_y, \sigma_y^2)$, where

$$\mu_y = \mu_1 - \mu_2 \quad (10)$$

$$\sigma_y^2 = \frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}. \quad (11)$$

The mean throughput of configuration C_i is $1/\mu_i$. TCC continues to add threads if the throughput ratio $\frac{1}{\mu_2}/\frac{1}{\mu_1} \geq 1 + q$, where q is the throughput improvement threshold. Considering (10), this is equivalent to $\mu_y \geq \mu'$, where $\mu' = \frac{q}{1+q}\mu_1$. (12)

We want to collect a minimum number of samples, $n = n_1 + n_2$, so that the variance σ_y^2 in (11) is small enough and we can state with high confidence either $\text{Prob}\{Y \geq \mu'\} \geq 1 - \alpha$ or $\text{Prob}\{Y < \mu'\} \geq 1 - \alpha$ holds. Here $1 - \alpha$ is the confidence level ($0 < \alpha < 0.5$). However, in the worst case when $\mu_y = \mu'$, both $\text{Prob}\{Y \geq \mu'\}$ and $\text{Prob}\{Y < \mu'\}$ are always 0.5, no matter how many samples we collect. This precludes us from deciding whether C_2 is significantly better than C_1 . We use an indifference zone $[L, H]$ to handle the case when $\mu_y \approx \mu'$.

$$L = (1 - \beta/2)\mu' \quad (13)$$

$$H = (1 + \beta/2)\mu' \quad (14)$$

Here β is a small constant, e.g., $\beta=0.1$. Now we want to collect just enough samples so that at least one of the two conditions below holds:

$$\text{Prob}\{Y \geq L\} \geq 1 - \alpha, \quad \text{or} \quad (15)$$

$$\text{Prob}\{Y \leq H\} \geq 1 - \alpha. \quad (16)$$

TCC adds more threads if only (15) holds, or if both (15) and (16) hold but $\text{Prob}\{Y \geq L\} \geq \text{Prob}\{Y \leq H\}$.

Let $Z \sim N(0, 1)$, and $\text{Prob}\{Z \leq Z_{1-\alpha}\} = 1 - \alpha$. Combining (15) and (16), we have

$$\sigma_y \leq \frac{1}{Z_{1-\alpha}} \max(H - \mu_y, \mu_y - L). \quad (17)$$

Combining (11) and (17), the problem of minimizing the total number of measurement samples can be formulated as the optimization problem below.

Minimize	$n = n_1 + n_2$
Subject to	
$\sigma_y^2 = \frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2} \leq \left\{ \frac{\max(H - \mu_y, \mu_y - L)}{Z_{1-\alpha}} \right\}^2 \quad (18)$	
$n_1, n_2 > 0 \quad (19)$	

$$\sigma_y^2 = \frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2} \leq \left\{ \frac{\max(H - \mu_y, \mu_y - L)}{Z_{1-\alpha}} \right\}^2 \quad (18)$$

$$n_1, n_2 > 0 \quad (19)$$

Solving this problem using Lagrange multipliers, we obtain the minimum number of samples we need:

$$\hat{n}_1 = \sigma_1(\sigma_1 + \sigma_2) \left\{ \frac{Z_{1-\alpha}}{\max(H - \mu_y, \mu_y - L)} \right\}^2 \quad (20)$$

$$\hat{n}_2 = \sigma_2(\sigma_1 + \sigma_2) \left\{ \frac{Z_{1-\alpha}}{\max(H - \mu_y, \mu_y - L)} \right\}^2 \quad (21)$$

Both \hat{n}_1 and \hat{n}_2 have the largest value when

$$\mu_y = \frac{H + L}{2} = \mu'. \quad (22)$$

When collecting samples for C_1 , we have no data for C_2 and hence μ_y is unknown. We have to make the conservative assumption in (22). As C_1 and C_2 are close, we assume $\sigma_1 \approx \sigma_2$. With these assumptions, (20) is simplified as (23) below. (Note that it is possible to run C_1 and C_2 back and forth in an interleaving fashion in order to accurately estimate μ_y rather than conservatively using (22) for μ_y , but this would complicate TCC's state machine in Figure 3.)

$$\hat{n}_1 = 8 \left(\frac{\sigma_1 Z_{1-\alpha}}{H - L} \right)^2. \quad (23)$$

Finally, combining (12), (13), (14), and (23), we have

$$\hat{n}_1 = 2 Z_{1-\alpha}^2 \left(\frac{1}{\beta} \right)^2 \left(1 + \frac{1}{q} \right)^2 \left(\frac{\sigma_1}{\mu_1} \right)^2. \quad (24)$$

The minimum number of samples for C_2 can be derived from (18) and (23):

$$\hat{n}_2 = \frac{(\sigma_2 Z_{1-\alpha})^2}{\{ \max(H - \mu_y, \mu_y - L) \}^2 - \frac{(H-L)^2}{8}}. \quad (25)$$

When collecting samples for C_2 , we have data for both C_1 and C_2 , and hence can estimate μ_y from (10).

3.4.2 Practical Issues

Our method does not rely on any assumption about the exact distribution of X_i , but needs to estimate the mean μ_i and variance σ_i^2 , as they are used in (24) and (25). TCC estimates them by taking n_i^* initial samples from configuration C_i , and then uses the sample mean μ_i^* and sample variance S_i^2 to replace μ_i and σ_i^2 . In practice, we observe that sometimes the event-processing engine experiences long pauses caused by extreme activities such as Java garbage collection or startup of a heavy external program. For example, on a fragmented large heap, Java garbage collection can take as long as 20 seconds.

These long pauses are not an inherent part of the variance in service time, but they make the calculated sample variance S_i^2 (and accordingly \hat{n}_i) unusually large. We address this issue by filtering out abnormally large samples. Empirically, we find that abnormal samples caused by long pauses are rare, and discarding the top 1% largest samples is sufficient to filter them out.

Another challenge is to handle the periodical, bulk-arrival pattern of IT events. After processing one block

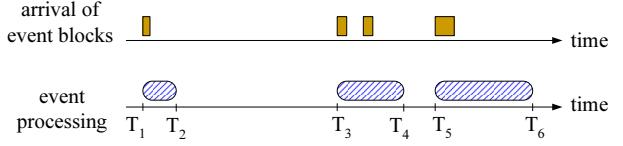


Figure 6: TCC excludes idle time from throughput calculation. Suppose n events are processed in this example. The throughput is calculated as $\frac{n}{(T_2-T_1)+(T_4-T_3)+(T_6-T_5)}$ instead of $\frac{n}{T_6-T_1}$. This method discounts the influence of an unstable event arrival rate and helps TCC operate robustly.

of events, an NCI server remains idle until the next block arrives. TCC excludes this idle time from throughput calculation (see Figure 6), because the low throughput in this case is caused by the lack of input events rather than by a sub-optimal thread configuration.

3.5 Selection of Parameter Values

Recall that TCC reduces threads in the base state by $w\%$, and then repeatedly add threads so long as every $p\%$ increase in threads improves throughput by $q\%$ or more. Now we put together the results in Sections 3.2, 3.3, and 3.4 to guide the selection of these parameters.

Equations (1) and (2) are the conditions for friendly resource sharing. Suppose p 's value is already determined. Using queueing models such as those in Figure 4, it can be shown that, relative to p , q should be as small as possible in order to achieve maximum throughput. Therefore, for a given p , we choose for q the smallest value allowed by (1). Once p and q are determined, we choose for w the smallest value allowed by (2), because a small w keeps more threads in the base state and allows TCC to finish an exploration cycle more quickly. Table 1 lists the appropriate values of q and w for different p .

p	10%	15%	20%	25%	30%	35%	40%	45%	50%
q	5.4%	8.2%	11.5%	14%	17%	20.5%	24%	27%	31%
w	28%	32%	33%	39%	42%	50%	56%	56%	63%

Table 1: Appropriate values of q and w for a given p .

The next step is to choose a configuration in Table 1. This table as well as (1) and (2) shows that both q and w increase as p increases. Equation (24) suggests that a large q is preferred, because it allows TCC to make a control decision with fewer measurement samples. On the other hand, we prefer a small w , as it keeps more threads in the base state. Moreover, we prefer a moderate p , because a large p has a higher risk of moving the system into severe thrashing in a single tuning step, whereas a small p may require many tuning steps to settle in a new steady state after a workload change. To strike a balance between all these requirements, we choose ($p=25\%$, $q=14\%$, $w=39\%$) as our default configuration.

In the remove-thread state, TCC repeatedly removes $r\%$ threads until the throughput becomes a certain fraction (e.g., 95%) of the maximum throughput achieved during a tuning cycle. The remove-thread state does fine tuning and we use $r=10\%$ by default.

4 Experimental Results

We have implemented TCC in Java and integrated it with IBM Tivoli Netcool/Impact [16]. We have evaluated TCC under a wide range of workloads. Experiments demonstrate that an NCI cluster is scalable, and TCC can handle various types of bottleneck resources. We also compare TCC with revised versions of TCP Vegas [4].

Unless otherwise noted, each machine used in the experiments has 5GB memory and two 2.33GHz Intel Xeon CPUs, running Linux 2.6.9. All the machines are hosted in an IBM BladeCenter, where the network round-trip time is only $90\mu\text{s}$. The network delay of a large enterprise's IT environment can be much longer, e.g., varying from 1ms to 100ms when the database with customer profiles is managed at a central remote site for security reasons. To evaluate the effect of network delay, in some experiments, the messages between two machines go through a software router that allows us to introduce message delays in a controlled manner.

4.1 NCI Cluster Scalability

Figure 7 shows the scalability of an NCI cluster running TCC, when executing an event-enrichment policy that is widely used in real deployments. The topology of this experiment is shown in Figure 4(c). The event source is Netcool/OMNIbus ObjectServer 7.1 [15]. We developed a tool to automatically feed IT events to ObjectServer, from which the master NCI server fetches events. The external database is MySQL 4.1.12. When processing one event, an NCI server does some local analysis, fetches service contextual information from MySQL, adds it into the event, and finally writes the enriched event back to ObjectServer. In this setup, MySQL caches all data in memory and NCI servers are the bottleneck.

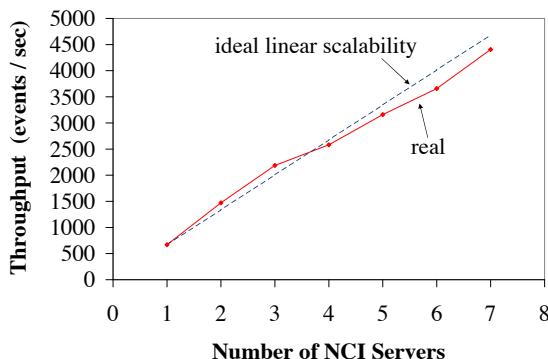


Figure 7: Scalability of NCI.

In Figure 7, the NCI cluster shows almost linear scalability, because the NCI servers fetch events in batches, and the slave NCI servers directly retrieve data from MySQL and write events back to ObjectServer without going the master. In addition, the linear scalability is also due to another algorithm we designed to dynamically regulate the event-fetching rate and event batch size so that the event-processing pipeline moves smoothly without stall. This feature is beyond the scope of this paper.

4.2 CPU Bottleneck

In the rest of the experiments, we study the detailed behavior of TCC. These experiments use the event-escalation topology in Figure 4(b), in which the NCI server processes an event and then invokes an external program through its HTTP/XML interface for further processing. Below, we simply refer to this external program as the “Web application” and its hosting machine as the “Web machine.” The IPL script executed by the NCI server is specially designed so that we can control its service time on the NCI server. Similarly, the service time of the Web application can also be controlled. Both service times follow a Pareto distribution $p(x) = k \frac{C^k}{x^{k+1}}$, where $k = 2.5$. We adjust C to control the service time.

Figure 8(a) shows the CPU utilization (“NCI-CPU” and “Web-CPU”) and the number of threads in an ex-

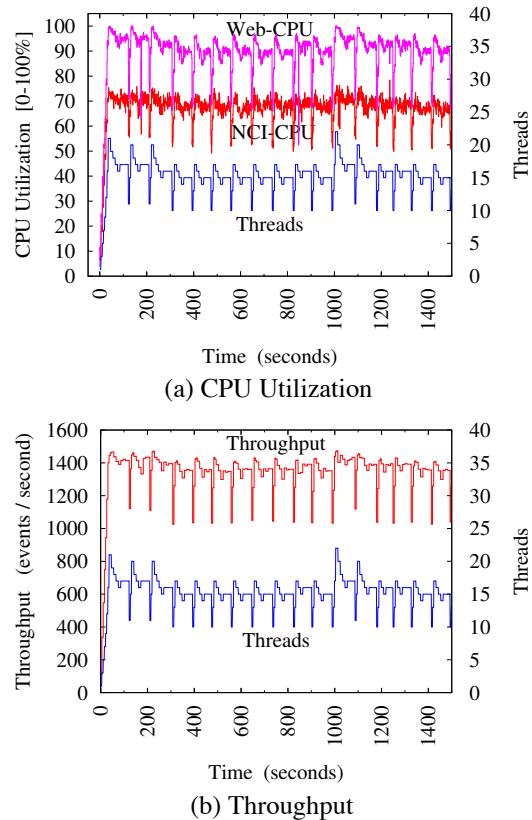


Figure 8: The Web machine's CPU is the bottleneck.

periment where one NCI server works with the Web application. The x -axis is the wall clock time since the experiment starts. In real deployments, after processing one block of events, the NCI server remains idle until the next block arrives. This experiment generates IT events in such a way that the NCI server never becomes idle. Otherwise, the CPU utilization would drop to 0 during repeated idle time, making the figure completely cluttered. We conducted separate experiments to verify that the idle time between event blocks does not change TCC’s behavior, because TCC excludes the idle time from throughput calculation (see Figure 6).

In this experiment, the mean service time is 1ms for NCI and 1.3ms for the Web application. (Note that the actual service times are random variables rather than constants.) Therefore, the Web machine is the bottleneck. The messages between the NCI server and the Web application go through the software router, which adds about 5ms delay in round trip time. The curves show periodical patterns. Each period is a complete tuning cycle during which TCC starts from the base state and eventually moves back to the steady state. In real deployments, TCC operates in the steady state for a relatively long period of time before it starts the next round of exploration. In this experiment, TCC is configured to stay in the steady state for only about 50 seconds. Otherwise, the curves would be mostly flat.

During the first tuning cycle in Figure 8(a), TCC exponentially increases the number of threads. At time 85 seconds, it moves into the steady state for the first time with 17 threads. During latter tuning cycles, the steady-state threads vary between 15 and 17. This oscillation is due to noisy measurement data. Regardless, TCC avoids saturating the bottleneck resource, and the Web machine’s CPU utilization stays around 90%.

Figure 8(b) shows event-processing throughput, which closely follows CPU utilization in Figure 8(a). This is because throughput is proportional to the utilization of the bottleneck resource (i.e., CPU in this experiment). Due to space limitation, below we omit throughput figures and focus on bottleneck resource utilization.

4.3 Memory Bottleneck

The experiment in Figure 9 evaluates how TCC works with memory bottleneck and how it recovers from memory thrashing. This experiment uses machines with relatively more CPUs and less memory in order to trigger memory thrashing—each machine has eight 3.2GHz Intel Xeon CPUs and 1GB memory. The mean service time is 8ms for NCI and 1ms for the Web application. The message delay is set to 50ms. Initially, the NCI server’s CPU is the bottleneck, and TCC uses 69 threads in the steady state to drive its utilization to 95%.

At time 496 seconds, the NCI server starts to invoke another API of the Web application, which consumes

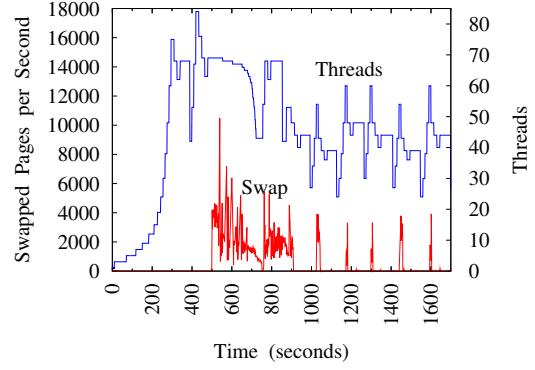


Figure 9: Memory bottleneck and memory thrashing.

a large amount of memory on the Web machine. In total, 69 concurrent threads consume more than 1GB physical memory and immediately drive the Web machine into memory thrashing. Figure 9 reports the Web machine’s page swaps monitored through `/proc/vmstat`. At time 496 seconds, the free memory drops sharply from 934MB to 19MB, page swaps increase from 0 to 4,000 pages/second, and the event-processing throughput drops from 1,011 events/second to 58 events/second. TCC detects this radical throughput change and restarts thread exploration. By time 945 seconds, TCC reduces steady-state threads down to 44. The Web machine becomes completely free of page swapping, its free memory rises to 106MB, and the throughput increases to 625 events/second. The tuning cycles are relatively long because the throughput is extremely low during memory thrashing and TCC needs time to collect samples.

After time 1,000 seconds, when TCC periodically explores new thread configurations, it increases the number of threads beyond 50, and causes page swapping to happen again (see the repeated spikes on the “Swap” curve after time 1,000 seconds). TCC observes that adding threads actually decreases throughput. It then removes threads and avoids thrashing. This experiment demonstrates that TCC can not only recover from memory thrashing but also avoids moving into thrashing.

4.4 Disk Bottleneck

The experiment in Figure 10 evaluates how TCC works with a disk bottleneck. Each machine used in this experiment has eight CPUs. The mean service time is 1ms for NCI and 2ms for the Web application. The message delay is set to 20ms. Initially, the Web machine’s CPU is the bottleneck, and TCC uses 107 threads in the steady state to drive its utilization to 95%.

At time 247 seconds, the NCI server starts to invoke another API of the Web application, which performs random search in a 60GB on-disk database. Now the bottleneck shifts to the Web machine’s disk. The Web machine’s CPU utilization drops from 95% to 5%, while

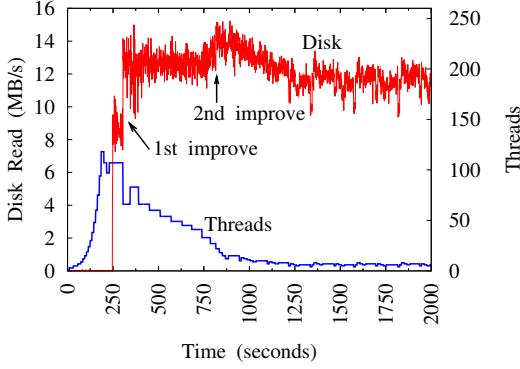


Figure 10: The Web machine’s disk is the bottleneck. Removing threads actually improves disk throughput.

the amount of data read from disk (monitored through `/proc/diskstats`) increases from 0 to 9MB/s. At time 305 seconds, TCC enters the base state and reduces threads from 107 to 66. With fewer threads, the disk throughput actually increases from 9MB/s to 12.5MB/s (“*1st improve*” in the figure). Note that the disk throughput is proportional to the event-processing throughput. When TCC reduces the number of threads down to 22 at time 816 seconds, the disk throughput further increases to 14MB/s (“*2nd improve*” in the figure). TCC continues to remove threads to avoid saturation. Eventually it stabilizes around 7 threads in the steady state, and the Web machine’s CPU utilization is only 1.5%. This experiment demonstrates that TCC can radically remove unnecessary threads (from 107 down to 7) when disk is the bottleneck, and disk can actually achieve higher throughput with fewer threads.

4.5 Network Bottleneck

For the experiment in Figure 11, the NCI server exchanges a large amount of data with the Web application when processing events. The mean service time is 1.5ms for NCI and 1ms for the Web application. There is no extra message delay between machines. The NCI server has higher CPU utilization than the Web machine, but the bottleneck of the whole system is network. Even if CPUs are still underutilized, TCC stops adding threads when the network bandwidth utilization reaches around 92%. Note that TCC works by observing changes in event-processing throughput, without even knowing which resource is actually the bottleneck.

4.6 NCI Working with a Competing Program

The experiment in Figure 12 evaluates TCC’s response to an external program that competes for the bottleneck resource. The mean service time is 1ms for NCI and 1.3ms for the Web application. The Web machine is the bottleneck. The message delay is set to 5ms. At time 286 seconds, an external program is started on the Web ma-

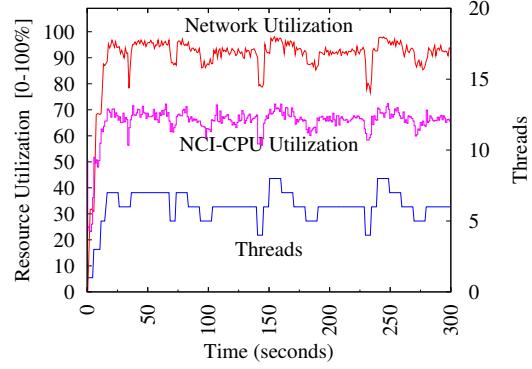


Figure 11: Network bandwidth is the bottleneck.

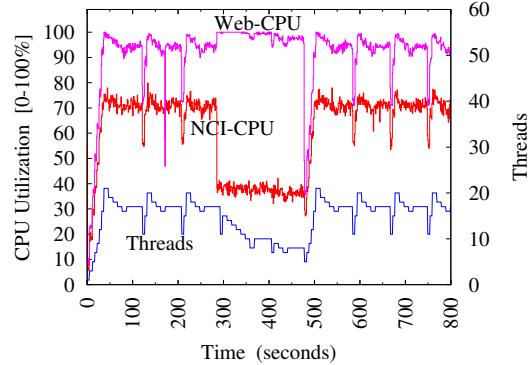
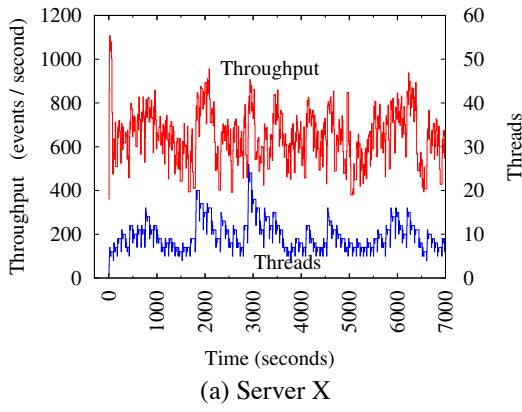


Figure 12: An external program competes for the bottleneck resource, which is the Web machine’s CPU.

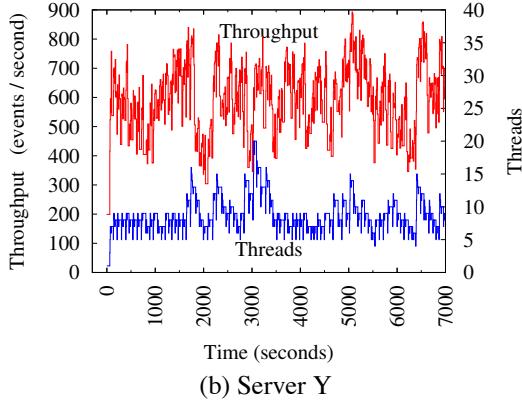
chine to compete for CPU. During the tuning cycle between time 293 and 368 seconds, TCC reduces steady-state threads from 17 to 10, and the Web machine is relieved of 100% saturation starting from time 341 seconds. The tuning cycle between time 406 and 441 seconds further reduces steady-state threads from 10 to 8. The external program terminates at time 477 seconds, and the Web machine’s CPU utilization drops sharply from 95% to 49%. During the following tuning cycle between time 483 and 553 seconds, TCC quickly increases steady-state threads from 8 to 17, and drives the Web machine’s CPU utilization back to 95%. This experiment demonstrates that TCC shares resources with a competing program in a friendly manner, and responds quickly when the bottleneck’s available capacity increases.

4.7 Two NCI Servers Sharing a Bottleneck

The experiment in Figure 13 runs two NCI servers to share the Web application. The mean service time is 1ms for NCI and 1.5ms for the Web application. The Web application is the shared bottleneck that limits the throughput of both NCI servers. This experiment introduces no extra message delays between machines. Server X starts first and quickly drives the throughput to as high as 1,100



(a) Server X



(b) Server Y

Figure 13: Two competing NCI servers work with the Web application. The latter is the shared bottleneck.

events/second (see the throughput spike around time 0). After server Y starts, X and Y share the bottleneck resource in a friendly manner. Their throughput oscillates around 550 events/second and their steady state oscillates around 8 threads. Sometimes one server mistakenly increases its threads beyond its fair share (due to noisy measurement data), which causes the other server to also increase its threads in order to get its fair share (see the thread spikes around time 3,000 seconds). However, the friendly resource sharing logic built in TCC ensures that the competition does not escalate, and they gradually reduce their threads back to the normal level.

4.8 Comparison of Different Controllers

To our knowledge, no existing controllers are designed to maximize the throughput of general distributed systems while not saturating the bottleneck resource. The closest to TCC is TCP Vegas’ congestion avoidance algorithm [4]. It computes the difference D between the actual throughput and the expected throughput, and increases the concurrency level if D is small, or decreases the concurrency level if D is large. The expected throughput is calculated as the product of the concurrency level and a baseline throughput. The baseline throughput is defined as the throughput achieved when

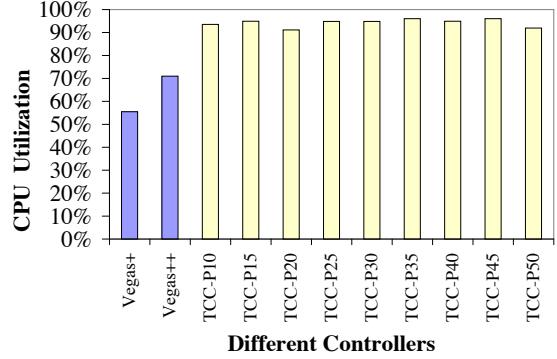


Figure 14: Comparison of different controllers.

the concurrency level is one and the network is completely free of cross traffic. A key challenge is to estimate the baseline throughput when cross traffic exists.

TCP Vegas is designed for and works well in the network domain, but it critically relies on an assumption that does not hold in general distributed systems—it assumes a processing unit’s service time is constant. Because of this assumption, TCP Vegas can use $\frac{1}{minRTT}$ to approximate the baseline throughput, where $minRTT$ is the minimum round trip time of packets. By contrast, the service time of a server in a general distributed system is inherently a stochastic process. For instance, the service time of a database query may vary over time, depending on the size of the database and the cache status. As a result, the actual baseline throughput for a general distributed system can be much lower than $\frac{1}{minRTT}$.

A complete redesign of TCP Vegas to work for general distributed systems would require (1) a method to estimate the baseline throughput when servers’ service times are stochastic processes and there exists cross traffic; (2) proving that the new algorithm shares resources with uncontrolled competing programs in a friendly manner; and (3) demonstrating that the new algorithm can achieve high resource utilization under typical topologies of distributed systems. The first task is especially challenging, and our initial study suggests that a satisfactory solution might not exist at all. We leave it as future work.

In Figure 14, “Vegas+” is our revised version of TCP Vegas. It runs in NCI at the application layer, and controls the number of event processing threads. It adjusts the concurrency level after measuring a stable throughput. In Vegas+, $minRTT$ is the minimum response time of event processing. In this experiment, the mean service time is 1ms for NCI and 2ms for the Web application. The network delay is set to 25ms. The Web machine is the bottleneck, and Vegas+ can only drives its CPU utilization to 55%. Vegas++ in Figure 14 is an enhanced version of Vegas+. It uses an accurate baseline throughput measured offline in a controlled environment that is free of cross traffic. Vegas++ is not a practical online algorithm, but we use it to study the potential of TCP

Vegas. Vegas++ improves the utilization of the bottleneck from 55% to 70%, which is still far from ideal. Vegas+ and Vegas++ use the parameters of TCP Vegas as recommended in [21]. The resource utilization may be improved by tuning these parameters, but the challenge is to prove that, with the new parameters, the algorithm is still friendly in resource sharing.

Vegas+ and Vegas++ are not full redesigns of TCP Vegas, as they do not solve the three challenging redesign tasks described above. Our initial study suggests that those tasks might not have a satisfactory solution at all. Here we use Vegas+ and Vegas++ to emphasize that the problem TCC solves is challenging, and no prior solutions can be easily adapted to solve the problem.

The other bars in Figure 14 show the performance of TCC under different configurations in Table 1 of Section 3.5. For instance, TCC-P25 is TCC’s default configuration ($p=25\%$, $q=14\%$, $w=39\%$), and TCC-P10 is ($p=10\%$, $q=5.4\%$, $w=28\%$). With the different configurations, TCC consistently drives the bottleneck resource utilization to 90-95%, showing that our guidelines in Section 3.5 for choosing TCC parameters are effective. Moreover, our guidelines ensure that TCC with these configurations is friendly in resource sharing.

5 Related Work

Performance control has been studied extensively for many applications, including Web server [28], search engine [2], storage [18], and scientific applications [25]. To our knowledge, no existing work uses a black-box approach to maximize the throughput of general distributed systems while trying to avoid saturating the bottleneck resource. TCP Vegas [4] is the closest to our algorithm, and a detailed discussion is provided in Section 4.8. Most existing algorithms [2, 23, 28] use a manually-configured and system-dependent response time threshold to guide performance control. If the threshold is set too high, the system will be fully saturated; if the threshold is set too low, the system will be underutilized.

We broadly classify existing controllers into four categories. Each category has an enormous body of related work, and we only review some representative ones.

The first category considers performance optimization as a search problem in a multi-dimensional parameter space. For instance, Active Harmony [25] uses the simplex method to perform the search. Existing methods of this category aggressively maximize performance without considering resource contention with an uncontrolled external program. Moreover, running multiple instances of the controller may result in severe resource saturation as each controller instance attempts to consume 100% of the shared bottleneck resource.

The second category uses classical control theory [13] to regulate performance. It requires the administrator to manually set a performance reference point. The system

then adjusts itself to stabilize around this reference point. If we apply this method to Netcool/Impact, the reference point would be achieving 90% bottleneck resource utilization. However, a straightforward implementation would require Netcool/Impact to monitor the resource consumptions of all third-party external programs working with Netcool/Impact, which is impractical in real deployments because of the diversity and proprietary nature of the third-party programs. Moreover, existing methods of this category are not sufficiently “black-box” and require information not available in Netcool/Impact deployment environments. For example, Triage [18] assumes knowledge of every resource-competing application’s service-level objectives, and the method in [24] assumes knowledge of every component’s performance characteristics obtained from offline profiling.

The third category uses queueing theory [12] to model a system with a fixed topology, and takes actions according to predictions given by the model. For instance, Pacifici et al. [23] use online profiling to train a machine-repairman model, which is used to guide flow control and service differentiation.

The fourth category includes various heuristic methods. SEDA [28] adjusts admission rate based on the difference between the 90-percentile response time and a manually-set target. Like TCP Vegas, Tri-S [27] is also designed for TCP congestion control and requires estimating a baseline throughput. MS Manners [9] regulates low-importance processes and allows them to run only if the system resources would be idle otherwise. It also needs to establish a baseline progress rate.

6 Conclusions

We presented TCC, a performance controller for high-volume non-interactive systems, where processing tasks are generated automatically in high volume by software tools rather than by interactive users, e.g., streaming event processing and index update in search engines. TCC takes a black-box approach to maximize throughput while trying to avoid saturating the bottleneck resource. We used analysis to guide the selection of its parameters, and designed a statistical method to minimize measurement samples needed for making control decisions. We implemented TCC and integrated it with IBM Tivoli Netcool/Impact [16]. Experiments demonstrate that TCC performs robustly under a wide range of workloads. TCC is flexible as it makes few assumptions about the operating environment. It may be applied to a large class of throughput-centric applications.

Acknowledgments

We thank Zhenghua Fu, Yaoping Ruan, other members of the Netcool/Impact team, the anonymous reviewers, and our shepherd Edward Lazowska for their valuable feedback.

References

- [1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS'02*, pages 1–16, 2002.
- [2] J. M. Blanquer, A. Batchelli, K. Schausser, and R. Wolski. Quorum: Flexible Quality of Service for Internet Services. In *NSDI'05*, pages 159–174, 2005.
- [3] L. Bouillon and J. Vanderdonckt. Retargeting of Web Pages to Other Computing Platforms with VAQUITA. In *The Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 339–348, 2002.
- [4] L. S. Brakmo, S. W. O’Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM'94*, pages 24–35, 1994.
- [5] J. Burrows. *Retail crime: prevention through crime analysis*. Home Office, 1988.
- [6] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187–200, 2000.
- [7] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *VLDB'00*, pages 200–209, 2000.
- [8] R. Collins, A. Lipton, T. Kanade, H. Fujiyoshi, D. Duggins, Y. Tsin, D. Tolliver, N. Enomoto, and O. Hasegawa. A System for Video Surveillance and Monitoring. Technical Report CMU-RI-TR-00-12, Robotics Institute, Carnegie Mellon University, 2000.
- [9] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *SOSP'99*, pages 47–260, 1999.
- [10] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *SOSP'97*, pages 78–91, 1997.
- [11] Google Reader. <http://www.google.com/reader>.
- [12] D. Gross and C. M. Harris. *Fundamentals of Queueing Theory*. John Wiley & Sons, Inc., 1998.
- [13] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Son, Inc., 2004.
- [14] Hewlett-Packard. Servicing the Animation Industry: HP’s Utility Rendering service Provides On-Demand Computing Resources, 2004. <http://www.hpl.hp.com/SE3D>.
- [15] IBM Tivoli Netcool Suite. <http://www.ibm.com/software/tivoli/welcome/micromuse/>.
- [16] IBM Tivoli Netcool/Impact. <http://www.ibm.com/software/tivoli/products/netcool-impact/>.
- [17] V. Jacobson. Congestion avoidance and control. In *SIGCOMM'88*, pages 314–329, 1988.
- [18] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage*, 1(4):457–480, November 2005.
- [19] G. Luo, C. Tang, and P. S. Yu. Resource-Adaptive Real-Time New Event Detection. In *SIGMOD'07*, pages 497–508, 2007.
- [20] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Int'l Workshop on Wireless Sensor Networks and Applications*, pages 88–97, 2002.
- [21] J. Mo, R. La, V. Anantharam, and J. Walrand. Analysis and comparison of TCP Reno and Vegas. In *INFOCOM'99*, pages 1556–1563, 1999.
- [22] B. Mobasher, R. Cooley, and J. Srivastava. Automatic personalization based on Web usage mining. *Communications of the ACM*, 43(8):142–151, 2000.
- [23] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinberg, A. Tantawi, and I. Whalley. Managing the Response Time for Multi-tiered Web Applications. Technical Report RC23942, IBM Research, 2006.
- [24] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, pages 289–302, 2007.
- [25] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *SuperComputing'02*, pages 1–11, 2002.
- [26] K. Thompson, G. Miller, and R. Wilder. Wide-area Internet traffic patterns and characteristics. *Network, IEEE*, 11(6):10–23, 1997.
- [27] Z. Wang and J. Crowcroft. A new congestion control scheme: slow start and search (Tri-S). *ACM SIGCOMM Computer Communication Review*, 21(1):32–43, 1991.
- [28] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *USITS'03*, pages 43–56, 2003.

Server Workload Analysis for Power Minimization using Consolidation

Akshat Verma Gargi Dasgupta Tapan Kumar Nayak Pradipta De Ravi Kothari

IBM India Research Lab

Abstract

Server consolidation has emerged as a promising technique to reduce the energy costs of a data center. In this work, we present the first detailed analysis of an enterprise server workload from the perspective of finding characteristics for consolidation. We observe significant potential for power savings if consolidation is performed using off-peak values for application demand. However, these savings come up with associated risks due to consolidation, particularly when the correlation between applications is not considered. We also investigate the stability in utilization trends for low-risk consolidation. Using the insights from the workload analysis, two new consolidation methods are designed that achieve significant power savings, while containing the performance risk of consolidation. We present an implementation of the methodologies in a consolidation planning tool and provide a comprehensive evaluation study of the proposed methodologies.

1 Introduction

According to an estimate [2] based on trends from American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE)[1], by 2014, Infrastructure and Energy (I&E) costs would contribute about 75% while IT would contribute a significantly smaller 25% towards the overall total cost of operating a data center. While there may be a difference in opinion on the relative proportion of I&E and IT costs, there is little disagreement that I&E costs would comfortably be the largest contributor to the cost of operating a data center. Reducing the I&E costs is, or will soon be, a major initiative of most data centers. One promising approach, prompted by virtualization and hardware-assisted isolation, for reducing the I&E costs is server consolidation.

Server consolidation is based on the observation that many enterprise servers do not maximally utilize the

available server resources all of the time. Co-locating applications, perhaps in individual virtual machines, thus allows for a reduction in the total number of physical servers, minimizes *server sprawl* as well as the total data center space requirements.

Consolidation reduces the total power consumed by the applications because existing servers are not energy-proportional, i.e., a significant amount of power is consumed even at low levels of utilization [26]. Though server features like *voltage* and *frequency scaling* modify this curve, there is still substantial power drawn at *idle* or *low utilization*. Consolidation thus provides an opportunity to reduce the overall power consumed by operating the servers in a range with a more attractive performance/Watt. For example, if two identical servers each utilizing, say 40% of the resources and drawing 80% of peak power were consolidated onto a single server, the consolidated server would be able to deliver identical performance at significantly less than the 160%(80 + 80) of the peak power. However, the key to effective consolidation is to estimate the (time-varying) resource requirements of individual applications (virtual machines) and to utilize these estimates along with the power profile of the physical servers to determine the consolidation strategy that can provide the best space-power benefits.

Server consolidation can be loosely broken into static, semi-static and dynamic consolidation. In static consolidation, applications (or virtual machines) are placed on physical servers for a long time period (e.g. months, years), and not migrated continuously in reaction to load changes. Semi-static refers to the mode of consolidating these applications on a daily or weekly basis. On the other hand, dynamic consolidation spans a couple of hours and requires a runtime placement manager to migrate virtual machines automatically in response to workload variations. Many virtualization vendors provide some tooling support for static consolidation [10, 15] with third party providers providing add-on features [9, 8] for inferring hardware constraints etc.

However, these tools essentially provide a policy-based framework with user defined policies and the placement intelligence is fairly simplistic. While multiple dynamic placement frameworks have been researched, in practise, administrators are often reluctant to migrate virtual machines automatically. Instead they prefer an offline or semi-offline framework, to evaluate the proposed placement and manually approve it. Hence, static and semi-static consolidation, where consolidation is performed daily or weekly is a much more appealing technique for administrators in real data centers. Though consolidation for minimizing server sprawl or power is not new, we are not aware of any prior study that utilizes correlation between workloads in a systematic way for determining the most effective static consolidation configuration.

1.1 Static Consolidation: What is new?

While dynamic workload placement has been a well studied problem, it assumes that there is minimal change in the resource requirement of the application during the (typically short) consolidation interval and hence a single resource size suffices. In the past, it has been assumed that the same assumption holds for static consolidation. However, for longer term consolidation there are significant reasons why this assumption fails. First, over a longer period of time, one is likely to see periods of peak as well as reduced application demand. Should the application size be taken to be the maximum, average or some other statistic? Second, placement decisions made based on historical data may not be accurate due to a systematic drift in the load. Third, there is an opportunity to utilize correlation between resource utilization on different virtual servers to influence the consolidation decision. Finally, long term placement has additional objectives like workload balance on active servers.

In summary, a static consolidation framework needs to deal with stochastic variables instead of fixed variables and the behavior of these variables need to be completely understood. We need to identify the right parameters to size workloads for medium or long intervals and assess their impact. It is also important to understand how correlation between applications can be employed for a more effective consolidation. The stability of various workload parameters need to be studied thoroughly to identify the risks involved in consolidation. Finally, existing placement methodologies need to be seen in light of the results of the workload characterization and should be modified, as needed.

1.2 Contribution

We present in this paper the first systematic server workload characterization of a large data center from the per-

spective of medium (semi-static) or long term (static) consolidation. We study the distribution of the utilization and occurrence of the peak utilization on servers relative to various percentiles and average metrics. We find that the tail of the distribution does not decay quickly for most servers implying that sizing applications based on average utilization has high degree of risk. We also observe significant correlation between applications hosted on different servers. We make the important observation that certain metrics like the 90-percentile as well as cross correlation between applications are fairly stable over time.

We use the insights obtained from our workload characterization to design two new consolidation methodologies, namely *Correlation Based Placement (CBP)* and *Peak Clustering based Placement (PCP)*. We implement the methodologies in a consolidation planning tool and evaluate the methodologies using traces from a live production data center. Our evaluation clearly establishes the superiority of the proposed algorithms. We also bring out the various scenarios in which each methodology is effective and show how to tune various parameters for different workloads.

The rest of the paper is organized in the following manner. We provide a background of server consolidation and the need for a system-level workload characterization in Sec. 2. A detailed workload characterization of a large data center is presented in Sec. 3. We use the insights from the workload characterization to design new placement methodologies in Sec. 4. We present an implementation and a careful evaluation of the proposed methodologies in Sec. 5. We conclude the paper with a summary of our key findings in Sec. 6.

2 Background

In this section, we first present a generalized formulation for server consolidation. The consolidation exercise can be formally stated as follows. Let there be N applications A_i that we need to place on M physical servers S_j for the period T . For each application A_i , let $C(A_i, t)$ denote the resource required in order to meet its SLA at time t . This paper does not deal with the problem of translating an application SLA to a resource value and assumes that $C(A_i, t)$ are available from monitored resource data. Let the capacity of a physical server S_j be denoted by $C(S_j)$ and X denote a specific consolidation configuration to specify the placement of applications on physical servers, i.e., an element of X , say $x_{ij} = 1$ if application A_i is placed on server S_j and 0 otherwise. Consolidation requires finding a configuration that optimizes a given cost function. For example, if the objective of consolidation is to optimize power, then we want to find a configuration X that minimizes $P(X)$, where $P(X)$

is a real valued function that provides the power consumed for a specific placement of applications. Further, the placement should ensure that the resource requirements are all applications are met for the entire duration T , i.e., $\forall t \in T, \sum_{i=1}^N x_{ij} C(A_i, t) \leq C(S_j)$. Further, we need to ensure that all applications are placed, i.e., $\sum_{j=1}^M x_{ij} = 1$.

Dynamic consolidation assumes that T is very short, leading to a single time-independent capacity demand $C(A_i)$ for each application. Hence, the capacity constraint is no longer stochastic in nature. In dynamic consolidation, for the estimation of $C(A_i)$ and $C(S_j)$, a popular metric in use is the RPE2 metric from IDEAS and almost all the commonly used servers are benchmarked with a fixed RPE2 value [24]. The RPE2 value of the server is used for $C(S_j)$ whereas the resource requirements of the application are estimated from the CPU utilization of the server. More specifically, if virtualization is not in use then the RPE2 of the host server multiplied by the maximum CPU utilization of the server in the period is used as an estimate of the resource requirements (size) of the application. If virtualization is in use, then the size is computed based on the entitlement of each virtual server on its host physical server, the CPU utilization of the virtual server, and the RPE2 of the host server.

Dynamic consolidation, due to its automated nature, is not preferred by data center administrators. Instead, they opt for static or semi-static consolidation strategies, where they can manually verify and approve the new configuration. However, for static or semi-static consolidation, the crux of the problem is to identify a size parameter that is useful for longer periods. Typically, an administrator may migrate virtual machines at the end of the day or on an identified day of the week. For such long durations, it is imperative to use a size that is able to save a lot of power (by consolidating on few power-efficient machines) as well as ensure that no SLA capacity violations would happen during periods of high load. Hence, the two important objectives in static consolidation are (i) *Overall Power Consumption* and (ii) *SLA Violation*, defined as number of time instances, when the capacity of server is less than the demand of all applications placed on it $\sum_{i=1}^N x_{ij} C(A_i, t) > C(S_j)$.

2.1 Related Work

Existing research in workload modeling can be classified into (a) aggregate workload characterization and (b) individual server utilization modeling. Aggregate workload characterization of a web server by Iyenger *et al.* [19] and workload models of a large scale server farm by Bent *et al* [3] fall in the first category. Individual server utilization has been studied in [5, 16, 4]. In [5], Bohrer *et al* use peak-trough analysis of commercial web servers to

establish that the average CPU utilization for typical web servers is fairly low. Similar observations on the peak-trough nature of enterprise workloads have been made in [16]. In [4], Bobroff *et al* perform trace analysis on commercial web servers and outline a method to identify the servers that are good candidates for dynamic placement. However, none of these studies provide a characterization of the inter-relationship between various workloads, as required for static consolidation.

There is also a large body of work on energy management in web clusters. Most of the cluster energy management literature addresses the problem of distributing requests in a web server cluster in such a way that the performance goals are met and the energy consumption is minimized [6, 21, 25, 17]. There are a number of papers that describe server or cluster level energy management using independent [22, 13] or cooperative DVS techniques [12, 18]. There are other efforts in reducing peak power requirements at server and rack level by doing dynamic budget allocation among sub-systems [14] or blades [23]. The work closest to the semi-static or static consolidation problem addressed in this paper are the dynamic consolidation methods proposed in [7, 26, 27, 4]. However, the relatively long duration for static consolidation introduces a stochastic nature to individual applications that is not captured in any of these frameworks.

3 Server Workload Analysis

We first present the details of the workload analyzed in this paper.

3.1 Trace Workload Details

The workload analyzed in this paper was collected from the production data center of a multi-national Fortune Global 500 company. The data center runs the core business applications of the enterprise as well as a service delivery portal. Each separate application suite was run from its own server cluster with a dedicated application team. Every application component in a suite ran from a dedicated virtual server, with many virtual servers hosted on a (typically) high end physical server. The traces were collected by the MDMS monitoring framework [20] deployed in the data center. The framework used its own sensors to collect CPU utilization for all the virtual servers with one entry every 5 minutes. We use traces collected over a 90 day period in the year 2007 for our analysis. We use the terms server and application interchangeably as each trace data corresponds to exactly one virtual server and application component.

The tracing methodology depends on sensors deployed in actual production servers over a long period. Hence, the data was noisy in parts due to routine system

Suite-Name	# of Servers	# of Days
AppSuite-1	10	19
AppSuite-2	18	13
AppSuite-3	13	25
AppSuite-4	16	37

Table 1: Workload Details for each cluster

maintenance (server reboots, performance troubleshooting that terminated all daemons including the sensors). Thus, the traces had missing or incorrect data for many time intervals during the trace period. We used a simple interval graph technique to identify the longest contiguous interval, where all the servers in one cluster had monitored data available. Hence, for each server cluster we identified a smaller period which had accurate monitored data available and used these smaller periods for our analysis.

The data center had a large number of clusters and we have selected 4 representative clusters (Table. 1) for this analysis. 'AppSuite-1', 'AppSuite-2' and 'AppSuite-4' had a 2 tiered application with application server components and DB server components. 'AppSuite-3' was a 3-tiered application suite with a few web servers, a few application servers, and a few DB servers. In most cases, multiple application servers used a common DB server. However, for 'AppSuite-2', few application servers had a dedicated DB servers assigned to them. There were no restrictions on co-locating two or more components of an application suite on the same physical server. The detailed information about the applications running in the data center and the virtual server to physical server mapping are withheld for privacy and business reasons.

3.2 Macro Workload Analysis

We begin our workload characterization study with server utilization of individual servers. Due to space limitations, we primarily report our observations on only one server cluster 'AppSuite-1', broadening our observations to other clusters only for important findings.

3.2.1 CPU Utilization Distribution

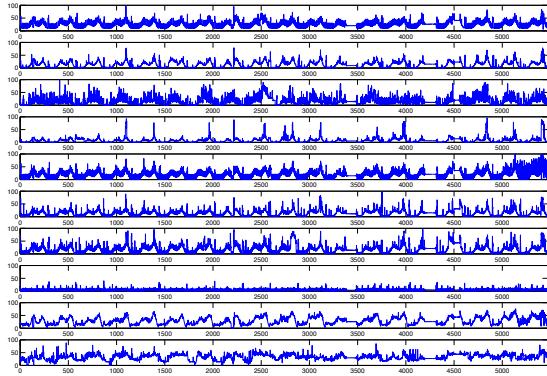


Figure 1: CPU Utilization for AppSuite-1 with Time

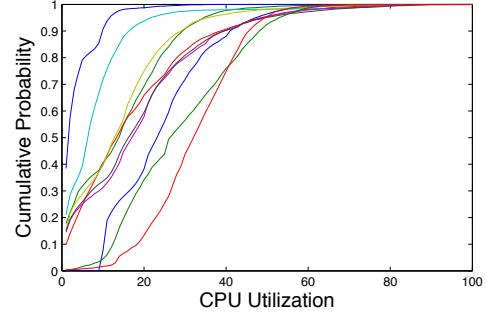


Figure 2: Cumulative Distribution Function of CPU Utilization for AppSuite-1

Fig. 1 shows the CPU utilization of each server in 'AppSuite-1', server1(top) through server10(bottom). The important observation to make here is that all servers barring server8 reach a CPU utilization of 100% at some point in time during the duration of the trace. This has an important implication for consolidation. **Observation 1:** *If consolidation is performed by reserving the maximum utilization for each application, the application may require capacity equal to the size of its current entitlement.* This observation is reinforced by taking a look at the Cumulative Probability Distribution (CDF) of CPU utilization (Fig. 2) for each server of 'AppSuite-1'. An interesting observation in the CDF plot however is the large skew in the distribution. For most of the applications, the CPU utilization at 90-percentile of the distribution is less than half of the peak CPU utilization. Such a skew can be utilized for a tighter consolidation by provisioning less than peak resource consumed by each application.

We drill down further into the skew of the CPU utilization distribution function in Fig. 3(a). We observe that the 99-percentile CPU utilization value is significantly less than the maximum CPU utilization in many cases. This is also in line with observations on other enterprise workloads made in [16]. Interestingly, the 90-percentile CPU utilization is about half or less of the maximum CPU utilization for 9 out of 10 servers. Interestingly, the gap between the 80 and 90-percentile values is less than 10% CPU utilization in all cases and less than 5% in many cases. We also look at the other server clusters in Fig. 3 and find the observations to hold there as well. However, in the 'AppSuite-2' cluster, a few servers have high utilization (Servers 15 to 18) for most of the interval. Hence, in these cases, both the 80 and 90-percentile values are reasonably close to the peak CPU utilization. The above findings lead us to our second important observation. **Observation 2:** *If we could size an application based on 90-percentile CPU utilization instead of maximum CPU utilization, it could lead to significant savings.*

We next observe the variability of the CPU utilization for different servers. To measure the variability, we computed the coefficient of variation (COV) for all the ap-

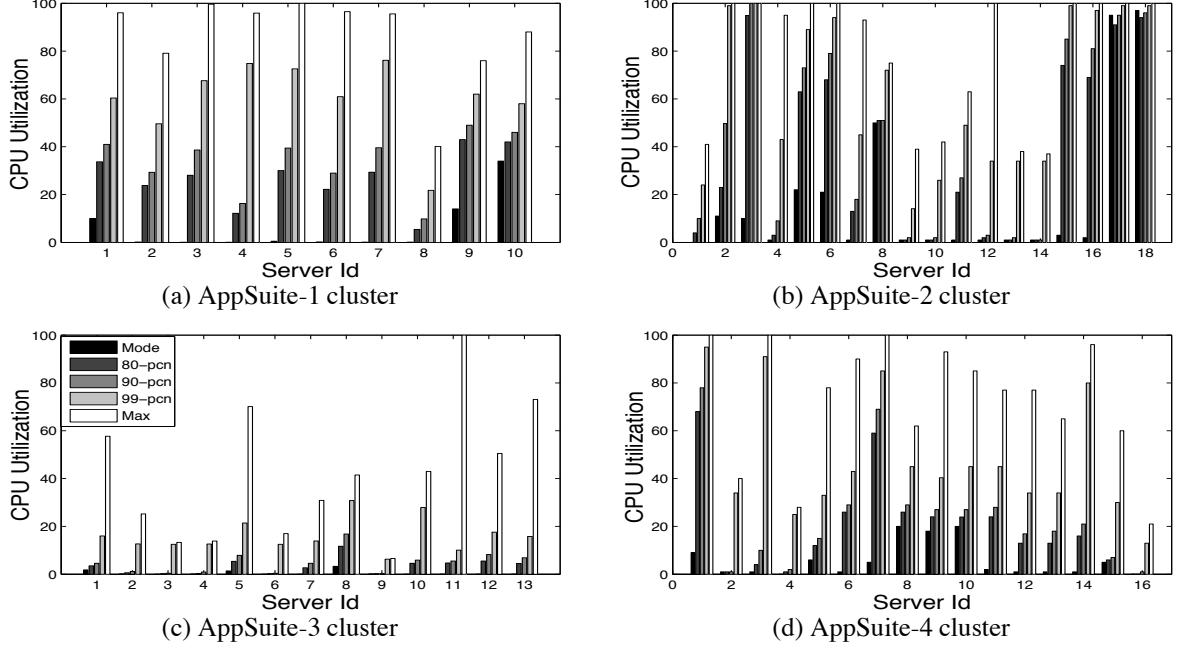


Figure 3: Comparison of Peak, Mode and Percentile CPU Utilization

plications in a cluster. The coefficient of variation is a normalized measure of dispersion of a probability distribution and is defined as $COV = \sigma/\mu$, where σ is the standard deviation and μ is the mean of the distribution.

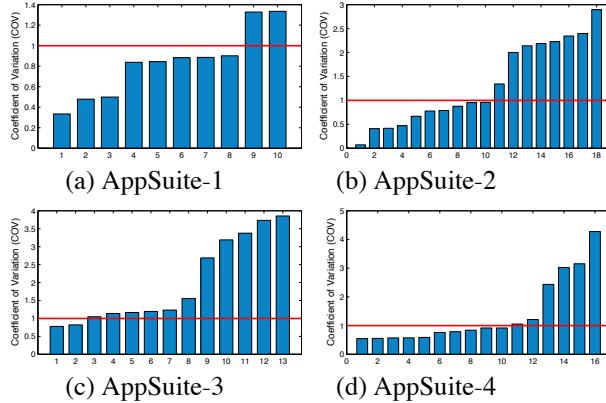


Figure 4: Coefficient of Variation for all clusters. Servers in each cluster are sorted by COV for easy comparison.

COV is a useful statistic for comparing the degree of variation and equals 1 for exponential distribution. Distributions with $COV>1$ (such as a hyper-exponential distribution) are considered high-variance, while those with $COV<1$ are considered low-variance. The coefficient of variations for all the clusters are shown in Fig. 4. We observe that all clusters have at least a few applications with high-variance distributions and 'AppSuite-3' has the largest number of applications with $COV>1$. There are also applications with low-variance distributions. However, it is well known that combining a heavy-

tailed distribution ($COV>1$) to another independent (or positively correlated) distribution with an exponentially decaying tail ($COV=1$) leads to an aggregate distribution, which is heavy-tailed. This leads to our third important observation. **Observation 3:** *If a statistical measure that ignores the tail of the distribution is used for sizing an application, the consolidated server may observe a large number of SLA capacity violations.*

3.2.2 Correlation

Our first few observations bring out the potential savings if applications were sized based on percentile values as opposed to peak values. However, sizing based on a non-peak value may lead to significant SLA violations if co-located applications peak together. Hence, we next study the correlation between applications belonging to the same application suite. The correlation between a pair of applications with timeseries $\{x_1, x_2, \dots, x_N\}$ and $\{y_1, y_2, \dots, y_N\}$ is represented by the *Pearson correlation coefficient*,

$$r_{xy} = \frac{N \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{N \sum x_i^2 - (\sum x_i)^2} \sqrt{N \sum y_i^2 - (\sum y_i)^2}} \quad (1)$$

Fig. 5 shows the pair-wise correlation between the applications of 'App-Suite1'. One may observe that there are many applications that have significant positive correlation. On the other hand, there are also a few applications (e.g., on Server 3, 8, and 10) that have minimal correlation with other applications. The observations highlight that (a) there is a risk of SLA violation if consolidation methods are not aware of correlation and (b) there is

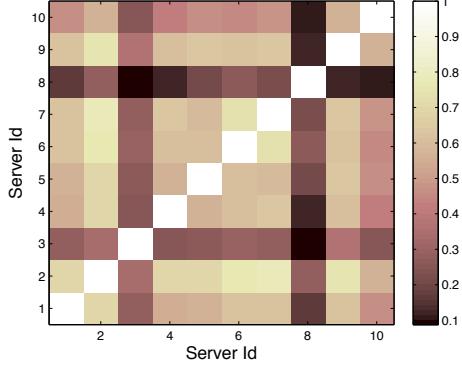


Figure 5: Inter-Server Correlation for AppSuite-1 cluster

potential for placing non-correlated applications to mitigate this risk. The other clusters have less correlation between servers but there are still a significant number of servers (more than 25%) that exhibit correlation with one or more other servers. One may observe that virtual servers that are part of a multi-component application have a high likelihood of being correlated. However, since in most cases, multiple (4 or 8) application servers were sharing a common DB server, the correlation was not strong. 'App-Suite2' however had 4 (application server, db server) pairs that were dedicated. As a result, even though the workload to this suite had low intrinsic correlation, the two-tier nature of the application suite introduced correlation. Hence, multi-tier applications with a one-to-one mapping between servers in different tiers are likely to exhibit correlation even for workloads with no intrinsic correlation. This leads to our next important observation. **Observation 4:** *There are both positively correlated and uncorrelated applications in a typical server cluster. Hence, correlation needs to be considered during placement to avoid SLA capacity violations.*

3.2.3 Temporal Distribution of Peaks

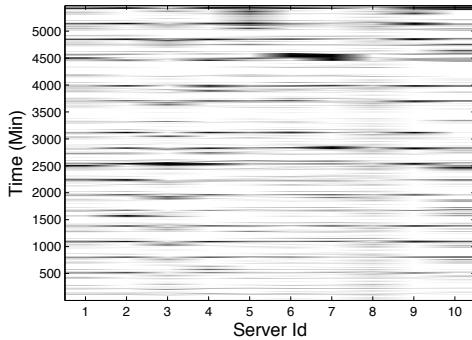


Figure 6: Duration of Peak CPU utilization ($> 90\text{-percentile}$) for AppSuite-1 cluster. Dark lines indicate sustained peaks.

We have used the correlation coefficient as an indicator of the temporal similarity between two applications. However, correlation is a comprehensive metric that captures temporal similarity between two applica-

tions at all levels (both peak and off peak). Capacity violations, though, occur when two applications sized by an off-peak value peak together. Hence, we look at the correlation between only the peaks for various applications in Fig. 6. We observe that there are apps with low correlation, but whose peaks may be correlated. Further, there also exists correlated apps whose peaks typically do not occur at the same time (e.g., Server 5 and 7). This leads to our next important observation. **Observation 5:** *Correlated Applications may not always peak together. Similarly, non-correlated applications may also peak together in some cases.*

3.3 Stability Analysis

Static and semi-static placement decisions are made for extended periods of time. Hence, there is a need to analyze the stability of workload statistical properties to ensure the reliability of the placement decisions. In this section, we study the workload periodicity, variation in statistical properties like mean, 90-percentile and correlation co-efficient over the observation period.

3.3.1 Periodicity analysis of utilization data

We first analyze the periodicity of the collected data. It will help to find the repeating patterns, such as the presence of a periodic signal which has been buried under noise. The usual method for deciding if a signal is periodic and then estimating its period is the auto-correlation function. For a discrete timeseries $\{x_1, x_2, \dots, x_N\}$ with mean μ and variance σ^2 , the auto-correlation function for any non-negative integer $k < N$ is given by

$$R(k) = \frac{1}{(N-k)\sigma^2} \sum_{n=1}^{N-k} [x_n - \mu] [x_{n+k} - \mu], \quad (2)$$

Essentially, the signal $\{x_n\}$ is being convolved with a time-lagged version of itself and the peaks in the auto-correlation indicate lag times at which the signal is relatively highly correlated with itself; these can be interpreted as periods at which the signal repeats. To enhance the analysis, we also computed the magnitude spectrum of the timeseries, $|X_k|$, where $\{X_k\}$ is the Discrete Fourier Transform (DFT) of $\{x_n\}$ and is defined by

$$X_k = \frac{1}{N} \sum_{n=1}^N x_n e^{-\frac{2\pi i}{N}(k-1)(n-1)}, \quad 1 \leq k \leq N. \quad (3)$$

We study the auto-correlation function and magnitude spectrum of the utilization data for all the applications and find that some servers exhibit nice periodic behavior, whereas some servers do not follow any particular pattern. Fig. 7 shows a periodic pattern with a time period of one day as the lag between two consecutive peaks in the auto-correlation function is one day and there is a peak in the magnitude spectrum corresponding to it.

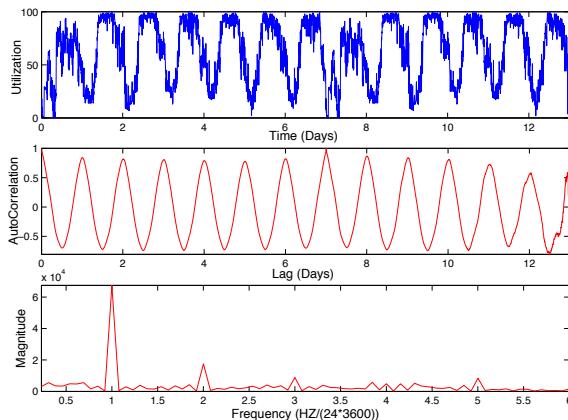


Figure 7: The timeseries, auto-correlation and frequency spectrum of this workload shows a periodicity of 1 day

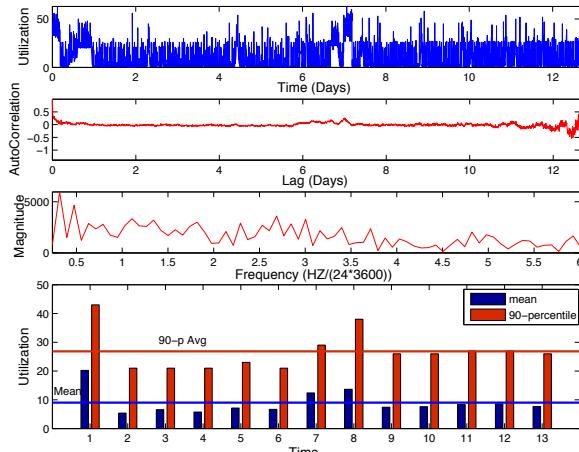


Figure 8: The timeseries, auto-correlation and frequency spectrum plot of the workload do not show any periodicity, but the mean and 90-percentile values show stability.

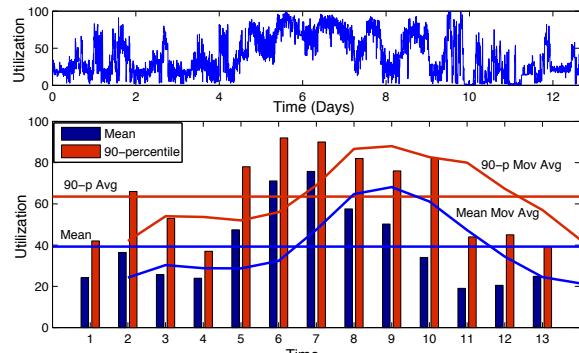


Figure 9: The timeseries has no regular pattern and the mean and 90-percentile statistics also vary significantly over the time period, but the moving averages track the statistic well.

This kind of workloads can be predicted with significant reliability. Many applications do not show any periodic pattern in the observed period, however, the statistical properties remain consistent over a long period. To analyze the consistency, we computed the mean and 90-percentile statistics over several windows of length 1 day. Fig. 8 shows that although the workload has no periodic pattern, the mean and 90-percentile statistics remains stable over most part of the observed period. Hence, for such workloads, the statistics can be estimated reliably. A third category of applications neither show any periodic behavior, nor any statistical consistency over a long period. However, for these applications, the moving averages follows the actual mean and 90-percentiles closely over the observed period (Fig. 9) and can be used for estimation. These observations lead to the following conclusion. **Observation 6:** *Some servers exhibit periodic behavior and the future pattern can be reliably forecasted with a day or a week of data. For many non-periodic servers, the statistical properties are fairly stable over time. For highly variable servers, an adaptive prediction method like MovingAverage should be used to estimate the statistical properties.*

3.3.2 Stability in Correlation

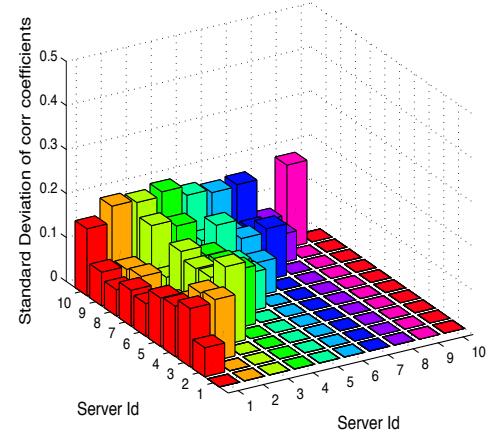


Figure 10: Stability of Correlation for App-Suite1 (Half the values have been deleted for visual clarity)

We have observed that correlation between applications should be used while making placement decisions. We next study the stability in correlation for AppSuite-1, which has the highest correlation amongst all the clusters. For this purpose, we compute the correlation between all pairs of applications for every day separately during the whole duration of the trace and compute the standard deviation across these daily correlation values. We observe in Fig. 10 that the standard deviation is fairly low, indicating the stability of correlation across time. **Observation 7:** *The correlation between the CPU utilization of various servers is fairly stable across time.*

4 Placement Methodologies

We now present various placement methodologies.

4.1 Workload-unaware Placement

We presented the *pMapper* power-aware application placement methodology and system in [26] in the context of a runtime placement controller. *pMapper* minimizes fragmentation using an enhancement of *First Fit Decreasing (FFD)* bin-packing algorithm and uses a novel *Order Preservation* property to select the right server for any application being migrated in order to minimize power. The algorithm optimizes the use of one resource (typically CPU utilization) during packing and treats other resources (e.g., memory, I/O) as constraints. Hence, it always comes up with a feasible packing for all resources but allocates only one resource in an optimized manner. The methodology used by *pMapper* does not focus on resource sizing of each VM for the next placement interval, which is predicted by a *Performance Manager*. An important thing to note here is that *pMapper* is designed for short consolidation intervals. There are two important implications of such an approach. Firstly, each application is sized independently and a single number is used to size an application. Secondly, as placement decisions need to be aware of application migration costs, few applications are migrated and the relocation decision takes an existing (old) placement into account. However, such an approach can still be applied for *static* consolidation with much longer consolidation intervals. In such a static placement scenario as the one considered in this paper, the *pMapper* methodology is naturally adapted by sizing an application based on the peak resource usage of the application in the (longer) placement period. Note further that in the case of SLA governed data centers, one can use less strict sizing functions. For example, if the SLA requires that resource requirements are met for at least 99% of the time, one could use a *VM* size that ensures a tail violation probability of 1%. Similarly, one may also choose to size all applications based on a metric like mode or median, if short-term violations are acceptable. We term this family of placement methodologies using peak, percentile, mode etc. based sizing as *Existing* placement methodologies.

4.2 Correlation Based Placement

We now present our first methodology that leverages the observations made from our workload analysis to place applications in a more effective manner. This methodology aptly named as the *Correlation Based Placement (CBP)* is based on the following important observations.

- The peak resource usage of an application is significantly higher than the resource usage at most other

times (e.g., size at 90% cdf). (Fig. 2, 3)

- If we size applications by an off-peak metric and place correlated applications together, there is a high risk of SLA capacity violation.
- If two uncorrelated applications are placed together and sized individually for a violation probability of $X\%$, the probability that both of them would violate their sizes at the same time is $(X^2)\%$.

To take an example, consider two applications A_1 and A_2 . Assume that both A_1 and A_2 have a maximum size of 1000 RPE2 units with a 90 percentile value of 500 RPE2 units. Further, assume that A_1 and A_2 are uncorrelated with each other. It is now easy to see that if we place A_1 and A_2 on a single server and allocate 500 RPE2 units each to both the applications, the probability that both of them would exceed their allocation at the same time is only 1%. Hence, provisioning based on 90 percentile and placing uncorrelated applications together can lead to a potential savings of 50% over the peak-based sizing method. *CBP* uses exactly these ideas to size individual applications based on a tail bound instead of the maximum size of the application. Further, it adds co-location constraints between positively correlated applications so that two such applications are not placed on the same server. The number of actual constraints added can be controlled using a tunable *Correlation Cutoff*. Hence, *CBP* proceeds in very similar manner to the *pMapper* algorithm with few key differences: (i) We add co-location constraints between any two positively correlated application pairs (A_i, A'_i) that exhibit a correlation coefficient above the *correlation threshold* (ii) We size applications based on a tail bound instead of the maximum value and (iii) In the inner loop of *pMapper* where we find the most power-efficient server S_j that has resources for an application A_i , we also make a check if none of the already placed applications on S_j have a co-location constraint with A_i . If indeed there is such an application, we mark the server ineligible and consider the next server for placing the application.

It is easy to see that *CBP* incurs an overhead in the computation of correlation for all application pairs.

Theorem 1 Given N applications and a timeseries with d points, *CBP* takes $O(N^2d)$ time to find the new placement.

We have proposed the *CBP* methodology that takes an existing dynamic consolidation algorithm and adapts it to work in a static or semi-static consolidation scenario. *CBP* adds co-location constraints between correlated applications to ensure that an application can be sized based on an off-peak value. However, it adds a hard constraint between correlated applications.

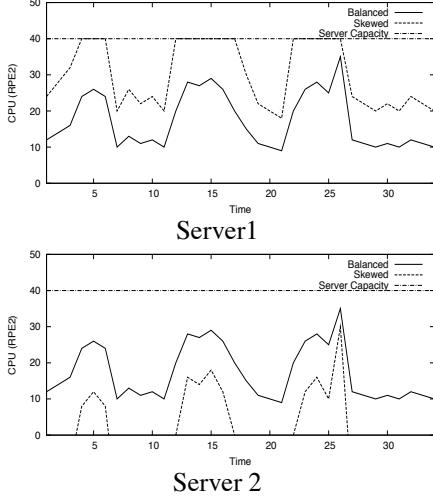


Figure 11: Fractional Optimal Solutions: Balanced and Skewed

We now take a closer look at the problem to understand the nature of the optimal solution. Consider a set of 6 applications that need to be placed on a set of servers, each with a capacity of 40. There are two potentially fractional optimal solutions, as shown in Fig. 11. A balanced solution would pack half of the timeseries in the first server and the other (balanced) half in the other server. A skewed solution would pack the first server to the maximum capacity and pack the remaining applications in the second server. *CBP* and other dynamic consolidation algorithms aim to approach the skewed optimal solution. However, from an administrative point of view it may be preferred to have balanced workload across all active servers.

A second problem with *CBP* may arise when there are many correlated applications. In the above example, if there are 3 applications that are positively correlated, we would need a minimum of 3 servers to satisfy the co-location constraints. Finally, computing the correlation between all pairs of applications is expensive (quadratic in nature) and may not be applicable for large number of applications and trace periods. We address these issues in another methodology next.

4.3 Peak Clustering Based Placement

We address the issues with *CBP* in a new consolidation method called *Peak Clustering based Placement (PCP)*. *PCP* is based on the following fundamental observations 1) Correlation between peaks of applications is more important than correlation across the complete timeseries of the applications. 2) A set of applications that peak together are distributed evenly across all active servers in the optimal solution. However, two applications with correlated peaks may still be co-located. 3) Co-located applications that do peak together can use a common buffer for their peaks and each have a reserva-

tion equal to an off peak value.

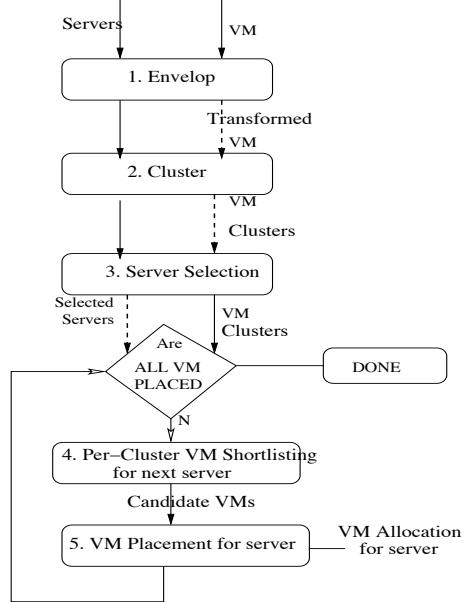


Figure 12: Overall Placement Flow

PCP uses these ideas to first identify clusters of applications with correlated peaks. One may observe that the number of such clusters may become very large if we use the original timeseries with the complete range of values. Hence, *PCP* uses a novel two-level envelop of the original time-series of each application for clustering. The envelop has a single value to represent all CPU utilization for the body of the distribution and another value for all points in the tail of the distribution. On each active server, it then reserves space for each cluster in proportion to the size (lower level of envelop) of the applications in that cluster and keeps a buffer space equal to the maximum peak across all clusters. Each application cluster shortlists a set of applications for its reservation and *PCP* does a final selection of applications for the server. The overall flow of *PCP* is described in Fig. 12.

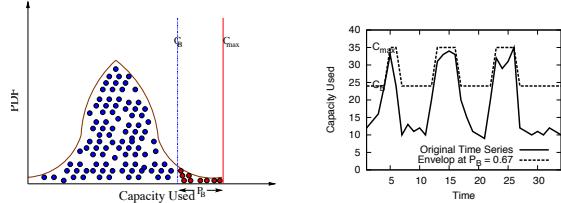


Figure 13: (a) Calculation of steps C_B, C_{max} for Envelop and (b) Envelop Creation

In step 1, *PCP* starts by using an *Envelop* function that transforms the original time series for each application to a two-level envelop. Given any tail bound P_B (e.g., 10%), the *Envelop* computes a value C_B such that the probability that application's CPU usage exceeds C_B is bounded by P_B . It also identifies the maximum capac-

ity used by the application as C_{max} (Fig. 13). We then transform the original timeseries by a two-level time series in the following manner. If at a given time, the capacity used by the application is greater than C_B , we replace it with C_{max} . Otherwise, replace it with C_B . Hence, the body of the distribution is replaced by C_B and is referred to as size. The tail is replaced by C_{max} . The timeseries for the transformed VM is stored as a set of ranges during which the size C_B is exceeded. The next step in *PCP* is to cluster workloads based on the correlation of their peak ranges. The clustering step uses a similarity function to identify workloads with correlated peaks. For each application A_i , the similarity function is used to identify if the envelop of the application is covered by any existing cluster center. If no such cluster center exists, then a new cluster is started with A_i as the cluster center.

Step 3 in the overall flow of *PCP* is to sort servers based on their power efficiency. We define marginal power efficiency for a server with capacity Cap_j running at capacity ρ_j as the slope of the capacity vs power curve at ρ_j capacity and overall power efficiency of the server as the ratio of the capacity of the server Cap_j to the peak power consumed by the server. The server selection method in our earlier work [26] used marginal power efficiency as the metric to sort servers. Dynamic consolidation requires us to make changes in an incremental, online manner from an existing configuration and hence, marginal power efficiency is a good metric for server selection. On the other hand, static consolidation may involve multiple global changes and hence, we use overall power efficiency to rank servers in *PCP*.

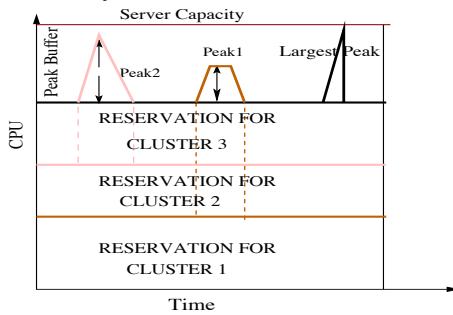


Figure 14: Server Reservation: Each cluster gets a proportional reservation. There is a buffer for the maximum peak across all clusters.

The final steps in *PCP* pack all the applications on the minimal number of servers, while packing the more power efficient servers first. The method picks the next highest ranked server and selects a set of applications from each cluster in a proportional manner. Given a server with capacity Cap_j to pack and a set of applications yet to be placed, we calculate the sum of the sizes (C_B) of all remaining applications as $TotalSize$. For each cluster M_k , we calculate the sum of the sizes $Size_k$

Analysis Period	120 hrs
Evaluation Period	24 hrs
P_B for PCP	0.9
Correlation Cutoff for CBP	0.5

Table 2: Baseline Parameters

of its applications and $Peak_k$ as the sum of the peak buffers ($C_{max} - C_B$) of its applications. We also calculate $MaxBuffer$ as the maximum of the peak buffers across all clusters. Once these metrics are computed, each cluster M_k selects a set of applications such that the overall size $CandSize_k$ and peak buffer $CandBuffer_k$ of the cluster is given by

$$CandSize_k = \frac{Size_k}{TotalSize + MaxBuffer} \quad (4)$$

$$CandBuffer_k \leq MaxBuffer \quad (5)$$

An example server reservation is described in Fig. 14. In this example, there are three clusters and a proportional reservation is made for each cluster that equals $CandSize_k$. Further, capacity is kept spare for peaks and equals the maximum peak across all the three clusters. Since the consolidation can only pick integral solutions, each cluster returns a set of applications whose sizes add up to its proportion or the last added application may exceed its proportion. Hence, as a final selection step, for each cluster that returns k candidates, we automatically select the first $(k - 1)$ candidates and add the last candidate to a tentative application pool. We then globally select the candidates from the tentative pool such that the capacity bounds of the server is not violated. In order to reduce fragmentation, at all levels of selection we break ties between applications by preferring the larger applications first. This allows *PCP* to strike a balance between reducing fragmentation costs and proportionally selecting applications across different clusters.

5 Experimental Evaluation

We have performed extensive experiments to evaluate the proposed methodologies. We first detail out our evaluation setup and then present some of our key findings.

5.1 Evaluation Setup

The methodologies have been implemented as part of an Consolidation Planning Tool from IBM called Emerald [11]. Emerald has built-in adapters to input trace data in various formats, a module to collect the current server inventory and a knowledge base that includes a catalog of various server platforms, their RPE2 values and power

models(Power Vs CPU Utilization). The power models are derived from actual measurements on the servers and are used by a *Power Estimation* module to estimate the power drawn by any candidate placement. We feed traces for the 4 application suites described in Sec. 3 for the evaluation study. Each server in this application suite was either a virtual machine or a standalone server in the data center. Hence, a single physical server may host one or more of these virtual servers in the data center. In our baseline setting, the physical servers were kept to be the same as in the data center. Further, Emerald allows an administrator to specify an analysis period for the traces followed by an evaluation period, where the effectiveness of the proposed placement is evaluated by the *Power Estimation* module. *PCP* uses a tail bound parameter P_B to create the envelop whereas *CBP* uses a correlation cutoff parameter to identify if two applications are correlated. The baseline settings of all experimental parameters are listed in Table. 2.

We evaluate the performance of the proposed methods in comparison with *Existing* methods based on dynamic consolidation. We run the *Existing* method with two different sizing functions: (i) *Peak_Existing* sizes all applications based on their peak values and (ii) *Mode_Existing* sizes all applications based on the mode of the distribution. There are three different objectives of consolidation: a)minimize power b) minimize risk of consolidation (c) balance workload across active servers. We use the number of capacity violations as the metric for risk. To investigate load imbalance, we estimate the average CPU utilization for all active servers during the evaluation period and identify the server with the maximum average CPU utilization. The difference between the CPU utilization of the highest loaded server and the average utilization across all servers is taken as the metric for load imbalance. We compare all the methodologies along all these objectives. We also measured the running time of various methodologies to assess their scalability.

In order to generalize our results and look at various settings, we also experimented with changes in our baseline setting. Since these methodologies need to deal with fragmentation, we also simulate a placement where the initial number of virtual servers on a physical server are increased or decreased. Towards this purpose, we assume that the virtual servers are placed on a different capacity server of the same platform, i.e., with more or less processors as required for the simulation. Further, seasonal variations were observed in many workloads and hence, we increase and decrease the training period from the baseline setting. Finally, we vary the tuning parameters of *CBP* and *PCP* and study their impact.

5.2 Performance Comparison

Power Consumed and SLA Violations: Fig. 15 shows

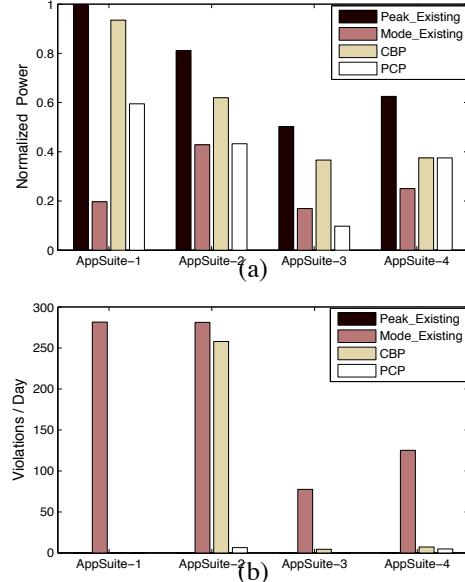


Figure 15: (a) Power Consumed and (b) SLA violations in all Clusters for various placement methodologies. Absence of bars indicate zero violation.

the power consumed in all 4 clusters as a result of placement recommendations made by *Peak_Existing*, *Mode_Existing*, *CBP* and *PCP* and the corresponding SLA violations. A striking observation is that the *Peak_Existing* methodology saves no power in AppSuite-1. Closer look at Fig. 3 reveals that barring one application, all other applications in this cluster have a peak CPU utilization of 80% or more. Hence, a methodology that sizes applications based on peaks is of no use in this cluster. The overall power consumed by *Mode_Existing* is typically the lowest as it sizes aggressively, while the *Peak_Existing* uses the maximum power. On the other hand, *Peak_Existing* has the lowest violations (typically zero) whereas *Mode_Existing* has the highest violations. Both *CBP* and *PCP* lie mid-way between these extremes, with *PCP* attaining about (20 – 40)% lower power consumption than *CBP* and significantly lower violations.

Another observation is that *CBP* saves significant power in comparison to *Peak_Existing* in all clusters other than AppSuite-1, while it faces very high violations for AppSuite-2. To understand this, we recall from Sec. 3 that the AppSuite-1 cluster has high correlation, AppSuite-2 has medium correlation and the remaining two clusters have a low correlation. As a result, *CBP* adds many co-location constraints in AppSuite-1 leading to a large number of active servers and not saving any power. In contrast, the medium correlation in AppSuite-2 results in *CBP* not making recommendations to separate these workloads. However, even though the workloads are not correlated, their peaks show up in the same time interval, leading to high violations by *CBP*. This peak correlation behavior is exploited only in the *PCP*

algorithm which decides to judiciously separate offending workloads if their simultaneous peaks can risk overshooting the server capacity, thereby causing violations. Thus PCP sees a nominal number of violations per hour in all clusters.

A surprising result observed is that *PCP* consumes less power than *Mode_Existing* for AppSuite-3. Recall that the server selection methodology in *Existing* methodology explores locally to reduce migration cost. On the other hand, server selection in *PCP* can explore new placements that entail a large number of migrations. Since AppSuite-3 was lightly loaded, consolidation would require large number of migrations. Hence, *PCP* came up with a different server selection than other methodology for this particular cluster, leading to additional power savings.

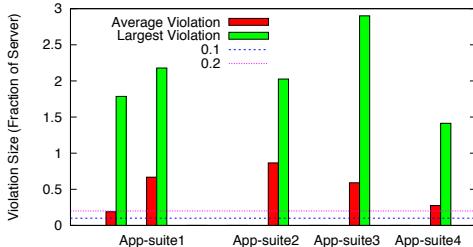


Figure 16: Violations for *Mode_Existing* on all active servers

We observed in Fig. 15 that *Mode_Existing* has the potential to save a lot of power but may incur a large number of violations. The correlation in the first two clusters of AppSuite-1 and AppSuite-2 especially impacts *Mode_Existing*, leading to very high violations. Results show that among the 9 active servers in all clusters, 5 of them faced SLA capacity violations. We study the size of the violations in the placement computed by *Mode_Existing* in Fig. 16. In practise, administrators allow a buffer capacity (typically 10% of the server) and hope that any peaks can be served from this buffer capacity. We observe that even a buffer capacity of 20% is not able to handle the burstiness of the workload. Amongst 5 servers, the average violation exceeds 20% of the server capacity in 4 servers and the peak violation exceeds the size of the server itself in all the servers. Hence, a workload-unaware strategy that uses a buffer capacity to handle peaks is not feasible in practise.

Workload Balancing: We next investigate the load balance across the active servers achieved by various methodologies in Fig. 17. We observe that *Mode_Existing* and *CBP* have servers that are always overloaded (average utilization of highest loaded server is 100%) for AppSuite-1 and AppSuite-2. Such a placement is not suitable in practise. Further, for all methodologies other than *PCP*, there is a large imbalance between the average utilization of the highest loaded server

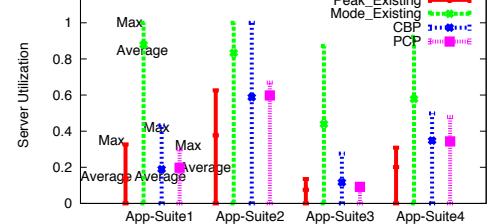


Figure 17: Average and Maximum Utilization of Servers

Cluster (No. of Apps)	Existing (ms)	PCP (ms)	CBP (ms)
AppSuite-1 (10)	10.1	47.7	34
AppSuite-3 (13)	13.5	55.2	55
AppSuite-4 (16)	30.1	39.8	81
AppSuite-2 (18)	21.2	47.7	107

Table 3: Running Time for various methodologies

and the average utilization of the cluster. This is a direct consequence of the design of algorithms, where *PCP* favors balancing and the other algorithms favor creating a skew.

Running Time: We study the running time of various methods in Table 3. The *CBP* algorithm very clearly says a super-linear relationship with the number of applications (N) because of the N^2 correlation co-efficient computation between all pairs of applications. The *Existing* method in general scales linearly with the increase in number of applications. *PCP* has a dependence on (a) the number of applications, (b) the number of peak ranges in a cluster and (c) the number of clusters it creates. Recapitulate that AppSuite-3 has the highest *CoV* (Fig. 4), which manifests in a large number of peak ranges. As a result, AppSuite-3 has the highest running time, even though the number of applications N is only 13. Overall, *PCP* has a running time that is double of *Existing* and fairly stable. The running time of *CBP* on the other hand increases super-linearly with the number of applications N .

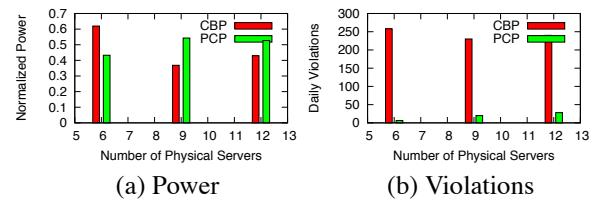


Figure 18: Impact of change in virtual servers per physical server in AppSuite-2

Fragmentation: We next investigate the ability of *CBP* and *PCP* to deal with fragmentation (ratio of application capacity $C(A_i)$ to server capacity $C(S_j)$). We change the original servers in the data center and simulate placement on a larger number of lower capacity servers. Fig. 18 shows the behavior of *CBP* and *PCP* with increase in number of physical servers for AppSuite-2, which has the largest number of virtual

servers. *CBP* adds a fixed number of co-location constraints and needs at least as many servers as the maximum number of constraints added to any application. Hence, it suffers when few high capacity servers are available. On the other hand, *PCP* tries to proportionally allocate applications from each cluster (of applications with correlated peaks) and hence should perform better when many applications can be packed on few high capacity servers. Both these intuitions are validated in Fig. 18 as *CBP* performs better with increase in number of servers and *PCP* fares worse. However, in both the cases, *CBP* suffers more violations than *PCP*. In summary, *CBP* is more suited when large applications are placed on low or medium capacity servers, whereas *PCP* is more suitable for consolidating a large number of applications on few high-end servers.

5.3 Tuning CBP

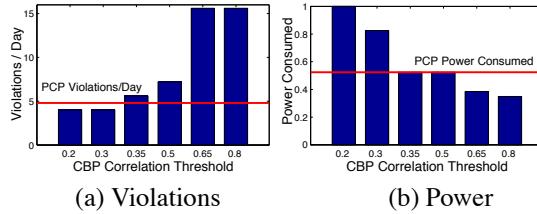


Figure 19: Power drawn and SLA Violations for *CBP* with changing correlation cutoff in AppSuite-4

The performance of *CBP* depends on the correlation cutoff parameter that is used to decide if correlation constraints need to be added between a pair of applications. Fig. 19 shows *CBP* performance with different thresholds, with the corresponding *PCP* metric shown as a reference line for AppSuite-4. Using a very low correlation threshold (0.2) creates constraints even between weakly correlated workloads thereby reducing the SLA violations (to even below that of *PCP*). However this comes at a cost of increasing the number of active servers, thereby consuming 50% more power than *PCP*. On the other hand, using a high correlation threshold (0.8) creates constraints only when workloads exhibit very high degree of correlation. As a result, the power consumed can be lowered below *PCP* at the cost of higher violations. We recommend an operating range (0.35 – 5) for significant power savings with reasonable number of violations. However one may set the threshold to 0.2 for critical applications and 0.8 for best-effort applications. We do observe that even though *CBP* can achieve either significant power savings or low violations, it is not able to achieve the trade-off as well as *PCP*.

5.4 Tuning PCP

The important configuration parameters for *PCP* are the length of the training period and the tail bound P_B used for creating envelopes. We first show the impact of

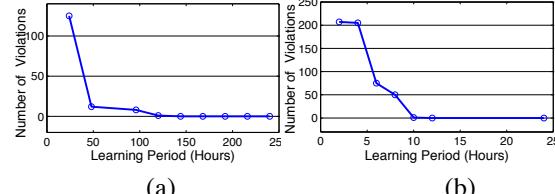


Figure 20: Performance of *PCP* with change in training period: (a)AppSuite-2 with weekly periodicity (b)AppSuite-4 with daily periodicity

available history (length of training period) on the performance of *PCP*. Fig. 20 shows SLA violations of AppSuite-2 and AppSuite-4 of *PCP* with change in the analysis period. We observe that without adequate history of workload repeatability, *PCP* can have very high number of violations. However, once adequate history is available, the number of violations fall to 0 for both the clusters. We have included AppSuite-2 and AppSuite-4 because the first cluster has a daily pattern whereas the second cluster has a weekly pattern. We observe that *PCP* is able to infer the characteristics of the workload in about half of the length of the period. Hence, for AppSuite-2, it takes about half a day of training data to reduce violations, whereas for AppSuite-4, we require about 4 days of training data. We also observe that the impact of historical trends happens in a discrete manner. Hence, for AppSuite-4, the availability of 2 full days of data leads to a significant decrease in violations as diurnal trends are available. However, any further data is not useful until we reach 5 days, when weekly trends become available. A key strength of *PCP* is that a user can easily determine the length of training period for a server cluster using a few sample runs and then store only the relevant history for future analysis. We next investigate

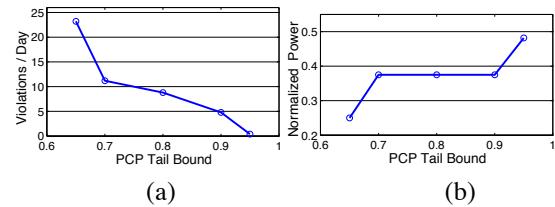


Figure 21: Performance of *PCP* with change in tail bound: (a) Hourly Violations (b) Power Consumed

the impact of the tail bound on the performance of *PCP* in Fig. 21. A high tail bound in *PCP* leads to a conservative size and smaller durations of peak activity. Hence, a high tail bound may lead to lesser violations but may lead to a higher power consumption. We observe this intuition holds in the experiments, as the violations fall to 0 for a tail bound of 0.95 but at the cost of higher power. Hence, the administrator can choose a bound based on the criticality of the applications in the cluster.

6 Conclusion

In this work, we have presented the server workload analysis of a large data center. We have investigated a large number of characteristics relevant for medium (semi-static) to long term (static) consolidation in order to save power. The workload study shows that there is a large potential for power savings by using off-peak metrics for sizing applications. However, correlation between applications can lead to significant capacity violations if consolidation methodologies do not take them into account. We design two new consolidation methodologies *CBP* and *PCP* that use an off-peak metric for sizing and another metric to ensure that peaks do not lead to violations. Our experimental evaluation shows that *PCP* achieves superior power savings, low violations and good load balance across active servers. Our work opens up further research in re-design of placement methods in light of the workload characteristics observed in our work.

7 Acknowledgements

We would like to thank our shepherd Mahadev Satyanarayanan and anonymous reviewers for insightful comments that have helped improve the paper.

References

- [1] ASHRAE Technical Committee 9.9. Datacom equipment power trends and cooling applications, 2005.
- [2] C. Belady. In the data center, power and cooling costs more than the it equipment it supports. <http://www.electronics-cooling.com/articles/2007/feb/a3/>, 2007.
- [3] L. Bent, M. Rabinovich, G. M. Voelker, and Z. Xiao. Characterization of a large web site population with implications for content delivery. In *WWW*, 2004.
- [4] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *IM*, 2007.
- [5] Pat Bohrer, Elmoottazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The case for power management in web servers. In *Proc. Power aware computing*, 2002.
- [6] J. Chase and R. Doyle. Balance of Power: Energy Management for Server Clusters. In *Proc. HotOS*, 2001.
- [7] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proc. ACM SOSP*, 2001.
- [8] Cloud computing software for data centers from Cassatt. <http://www.cassatt.com/products.htm>.
- [9] Server Consolidation and Virtualization Analysis by CiRBA. <http://www.cirba.com/>.
- [10] VMWare Guided Consolidation. <http://www.vmware.com/products/vi/vc/features.html>.
- [11] G. Dasgupta, A. Sharma, A. Verma, A. Neogi, and R. Kothari. Emerald: A tool to help data centers go green. In *Under Review*, 2008.
- [12] E. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proceedings of the Workshop on Power-Aware Computing Systems.*, 2002.
- [13] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, 2003.
- [14] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *Proc. of International Conference on Supercomputing*, 2005.
- [15] Virtual Iron: True Server Virtualization for Everyone. <http://www.virtualiron.com/>.
- [16] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. In *IISWC*, 2007.
- [17] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr., and R. Bianchini. Energy conservation in heterogeneous server clusters. In *PPoPP*, 2005.
- [18] Tibor Horvath. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Trans. Comput.*, 56(4), 2007.
- [19] A. K. Iyengar, M. S. Squillante, and L. Zhang. Analysis and characterization of large-scale web server access patterns and performance. In *Int'l World Wide Web Conference*, 1999.
- [20] Bharat Krishnamurthy, Anindya Neogi, Bikram Sengupta, and Raghavendra Singh. Data tagging architecture for system monitoring in dynamic environments. In *Proc. NOMS*, 2008.
- [21] K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *Proc. ISPASS*, 2003.
- [22] Karthick Rajamani, Heather Hanson, Juan Rubio, Soraya Ghiasi, and Freeman L. Rawson III. Application-aware power management. In *IISWC*, pages 39–48, 2006.
- [23] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. In *Proc. of International Symposium on Computer Architecture*, 2006.
- [24] About IDEAS Relative Performance Estimate 2 (RPE2). <http://www.ideasinternational.com/performance/>.
- [25] Cosmin Rusu, Alexandre Ferreira, Claudio Scordino, and Aaron Watson. Energy-efficient real-time heterogeneous server clusters. In *Proceedings of RTAS*, 2006.
- [26] A. Verma, P. Ahuja, and A. Neogi. pmapper: Power and migration cost aware application placement in virtualized servers. In *Proc. of ACM/IFIP/Usenix Middleware*, 2008.
- [27] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of hpc applications. In *Proc. of ACM ICS*, 2008.

RCB: A Simple and Practical Framework for Real-time Collaborative Browsing

Chuan Yue, Zi Chu, and Haining Wang

The College of William and Mary

{cyue,zichu,hnw}@cs.wm.edu

Abstract

Existing co-browsing solutions must use either a specific collaborative platform, a modified Web server, or a dedicated proxy to coordinate the browsing activities between Web users. In addition, these solutions usually require co-browsing participants to install special software on their computers. These requirements heavily impede the wide use of collaborative browsing over the Internet. In this paper, we propose a simple and practical framework for Real-time Collaborative Browsing (RCB). This RCB framework is a pure browser-based solution. It leverages the power of Ajax (Asynchronous JavaScript and XML) techniques and the end-user extensibility of modern Web browsers to support co-browsing. RCB enables real-time collaboration among Web users without the involvement of any third-party platforms, servers, or proxies. It allows users to perform fine-grained high quality co-browsing on arbitrary websites and webpages. We implemented the RCB framework in the Firefox Web browser and evaluated its performance and usability. Our evaluation results demonstrate that the proposed RCB is simple, practical, helpful and easy to use.

1 Introduction

Many end-user real-time applications have been widely used on the Internet. Real-time audio/video communication is enabled by voice/video over IP systems, real-time text-based communication is enabled by instant messaging systems, and real-time document sharing and collaboration is enabled by Web-based services such as Google Docs and Adobe Buzzword. However, one of the most popular Internet activities, Web browsing, is still heavily isolated. In other words, browsing regular webpages is still a process that is mainly between a user client and a remote Web server, and there is little real-time interaction between different users who are visiting the same webpages.

Collaborative browsing, also known as co-browsing, is the technique that allows different users to browse the same webpages in a simultaneous manner and collaboratively fulfill certain tasks. Co-browsing has a wide range of important applications. For example, instructors can illustrate online materials to distance learning students, business representatives can provide live online technical support to customers, and regular Web users can conduct online searching or shopping with friends.

Several approaches exist to achieve different levels of co-browsing. At one extreme, simple co-browsing can be performed by just sharing a URL in a browser's address bar via either instant messaging tools or Web browser add-ons (such as CoBrowse [5]) that are installed on each user's computer. URL sharing is lightweight, but it only enables very limited collaboration on a narrow scope of webpages. Collaboration is limited since users can only view webpages but cannot perform activities such as co-filling online forms or synchronizing mouse-pointer actions. Webpages eligible for this simple co-browsing method are also limited because: (1) most session-protected webpages cannot be accessed by just copying the URLs, and (2) in many dynamically-updated webpages such as Google Maps, the retrieved contents will be different even with the same URL.

At the other extreme, complex co-browsing can be achieved via screen or application sharing software such as Microsoft NetMeeting. To enable co-browsing activities, these solutions must grant the control of a whole screen or application to remote users. As a result, they place high demands on both security assurance and network bandwidth, and their use is more appropriate for some other collaborative applications than co-browsing.

A number of solutions have been proposed to support full functional co-browsing with moderate overhead. Based on the high-level architectures, these solutions can be classified into three categories: platform-based, server-based, and proxy-based solutions. Platform-based solutions build their co-browsing functionalities upon

specific real-time collaborative platforms [9, 11, 15, 30]. Server-based solutions modify Web servers to meet collaborative browsing requirements [2, 7, 13, 22, 28]. Proxy-based solutions use external proxies, which are deployed between Web servers and browsers, to facilitate collaborative browsing [1, 3, 4, 6, 12, 14]. However, as discussed in Section 2, the specific architectural requirements of these solutions limit their wide use in practice.

In this paper, we propose a simple and practical framework for Real-time Collaborative Browsing (RCB). The proposed RCB is a pure browser-based solution. It leverages the power of Ajax (Asynchronous JavaScript and XML) [20] techniques and the end-user extensibility of modern Web browsers to support co-browsing. RCB enables real-time collaboration among Web users without using any third-party platforms, servers, or proxies. The framework of RCB consists of two key components: one is RCB-Agent, which is a Web browser extension, and the other is Ajax-Snippet, which is a small piece of Ajax code that can be embedded within an HTML page and downloaded to a user’s regular browser. Installed on a user’s Web browser, RCB-Agent accepts TCP connections from other users’ browsers and processes both Ajax requests made by Ajax-Snippet and regular HTTP requests. RCB-Agent and Ajax-Snippet coordinate the co-browsing sessions and allow users to efficiently view and operate on the same webpages in a simultaneous manner.

The framework of RCB is simple, practical, and efficient. A user who wants to host a collaborative Web session only needs to install an RCB-Agent browser extension. Users who want to join a collaborative session just use their regular JavaScript-enabled Web browsers, and nothing needs to be installed or configured. End-user extensibility is an important feature supported by popular Web browsers such as Firefox [24] and Internet Explorer [26]. Thus, it is feasible to implement and run the RCB-Agent extension on these browsers. Meanwhile, currently 95% of Web users turn on JavaScript in their browsers [21], and all popular Web browsers support Ajax techniques [20]. As a result, joining a collaborative Web session is like using a regular browser to visit a regular website. The simplicity and practicability of RCB bring important usability advantages to co-browsing participants, especially in online training and customer support applications. RCB is also efficient because co-browsing participants are directly connected to the user who hosts the session, and there is no third-party involvement in the co-browsing activities.

Other distinctive features of RCB are summarized as follows. (1) *Ubiquitous co-browsing*: since no specific platform, server, or proxy is needed, co-browsing can be performed in many different places via any type of network connection such as Ethernet, Wi-Fi, and Bluetooth. (2) *Arbitrary co-browsing*: co-browsing can be

applied to almost all kinds of Web servers and webpages. Web contents hosted on HTTP or HTTPS Web servers can all be synchronized to co-browsing participants by RCB-Agent. Our RCB-Agent can also send cached contents including image and Cascading Style Sheets (CSS) files to participants, hence improving performance and accessibility of co-browsing in some environments. (3) *Fine-grained co-browsing*: co-browsed Web elements and coordinated user actions can be very fine-grained. Since RCB-Agent is designed as a browser extension, the seamless browser-integration enables RCB-Agent to fully control what webpage contents can be shared and what actions should be allowed to participants, leading to full functional high quality co-browsing.

We implemented the RCB framework in Firefox. As a browser extension, RCB-Agent is purely written in JavaScript. Ajax-Snippet is also written in JavaScript and it works on different browsers like Firefox and Internet Explorer. We evaluated the real-time performance of RCB through extensive experiments in LAN and WAN environments. Based on two real application scenarios (collaboratively shopping online and using Google Maps), we also conducted a formal usability study to evaluate the high quality co-browsing capabilities of RCB. Our evaluation results demonstrate that the proposed RCB is simple, practical, helpful and easy to use.

2 Related Work

The existing co-browsing solutions can be roughly classified into platform-based, server-based, and proxy-based solutions. Platform-based solutions build their co-browsing architectures upon special real-time collaborative platforms. As an early work in this category, GroupWeb [11] is built on top of the GroupKit groupware platform [18], and similarly GroupScape [9] is developed by using the Clock groupware development toolkit [10]. Two banking applications [15] for synchronous browser sharing between bank representatives and customers are designed on top of a multi-party, real-time collaborative platform named CollaborationFramework [19]. Recently, SamePlace [30] is built upon the XMPP (eXtensible Messaging & Presence Protocol) platform [32] to support co-browsing of rich Web contents. The strong dependence on specific collaborative platforms is the major drawback of these co-browsing solutions.

Server-based solutions modify Web servers and integrate collaborative components into servers to support co-browsing [2, 7, 13]. CWB (Collaborative Web Browsing) [7] is a typical example in this category. CWB consists of a controller module that runs on a Web server and a control panel that runs on a Web browser. The controller module is implemented as a Java servlet and is the central control point for collaborative activities. The con-

trol panel reports local browser instance changes to the controller module on the Web server, and it also polls the controller module for changes made by other users. In addition to CWB, some commercial software like Backbase Co-browse & Chat suite [22] and PageShare [28] also adopt this approach. However, these solutions have two obvious limitations: (1) they require Web developers to add controller modules to Web servers, and (2) the server-side modification is usually tailored and dedicated to individual websites, and it is infeasible to apply such a modification to most Web servers.

Proxy-based solutions rely on dedicated HTTP proxies to coordinate co-browsing among users [1, 3, 4, 6, 12, 14]. Users configure the proxy setting on their browsers to access the Internet via an HTTP proxy. The proxy serves co-browsing users by forwarding their HTTP requests to a Web server and returning identical HTML pages to them. The proxy also inserts applets (often in the form of Java applets [4, 12] or JavaScript snippets [3]) into the returned HTML pages to track and synchronize user actions. The major drawback of proxy-based solutions is the extra cost of setting up and maintaining such a proxy. Moreover, there are security and privacy concerns on using a proxy. Since all the HTTP requests and responses have to go through a proxy, each user has no choice but to trust the proxy.

3 Framework Design

In this section, we first present the architecture of the RCB framework. We then justify our design decisions. Finally, we analyze the co-browsing topologies and policies of RCB, and discuss the security design of RCB.

3.1 Architecture

The design philosophy of RCB is to make co-browsing simple, practical, and efficient. As shown in Figure 1, the architecture of the RCB framework consists of two major components. One is the RCB-Agent browser extension that can be seamlessly integrated into a Web browser. The other is Ajax-Snippet — a small piece of Ajax [20] code that can be embedded within an HTML page and downloaded to a user’s regular browser. For a user who wants to host a co-browsing session, the user (referred to as a *co-browsing host*) only needs to install an RCB-Agent browser extension. For a user who wants to join a co-browsing session, the user (referred to as a *co-browsing participant*) does not need to install anything and just uses a regular JavaScript-enabled Web browser. Our design philosophy of making the participant side as simple as possible is similar to the basic concept of many thin-client systems such as VNC (virtual network computing) [17].

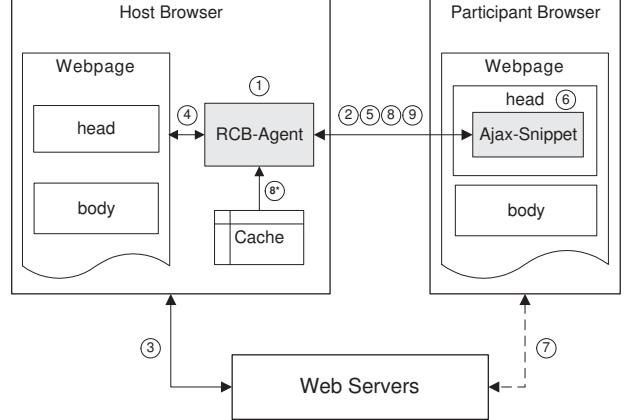


Figure 1: The architecture of the RCB framework.

In Figure 1, the *host browser* represents the browser used by a co-browsing host, and the *participant browser* corresponds to the browser used by a co-browsing participant. The *webpage* on each browser stands for a currently co-browsed HTML webpage. The displayed content of each webpage is the same on both browsers, but the source code of each webpage is different on the two browsers. The *cache* of the host browser is only read but not modified by RCB-Agent.

A co-browsing session can be broken down into nine steps. In step 1, a co-browsing host starts running RCB-Agent on the host browser with an open TCP port (e.g., 3000). In step 2, a co-browsing participant types the URL address of RCB-Agent (e.g., `http://example-address:3000`, where the example-address is a reachable hostname or IP address) into the address bar of the participant browser and sends a connection request to RCB-Agent. The RCB-Agent responds to a valid request by returning an *initial HTML page* that contains Ajax-Snippet. Ajax-Snippet will then periodically poll RCB-Agent, and the communication channel between the co-browsing host and participant is established.

On the host browser, whenever the co-browsing host visits a webpage (step 3), RCB-Agent monitors the internal browser-state changes and records file-downloading activities related to the webpage (step 4). When the webpage is loaded on the host browser, RCB-Agent creates an in-memory copy of the page’s HTML document and makes necessary modifications to this copy. Then, in step 5, upon receipt of a polling request from a participant browser, RCB-Agent will send the content of the modified copy to the participant browser.

On the participant browser, Ajax-Snippet will analyze the received content and replace the corresponding HTML elements of the current page, in which Ajax-Snippet always resides, with the received content (step 6). In addition to the HTML document that describes the page structure, a webpage often contains supplementary

objects such as stylesheets, images, and scripts. Therefore, to accurately render the same webpage, the participant browser needs to download all these supplementary objects. Based on RCB-Agent’s modifications on the copied HTML document, the RCB framework allows a participant browser to download these supplementary objects either from the original Web server (step 7), or directly from the host browser (step 8 and 8*).

Allowing a participant browser to directly download cached objects from the host browser can bring two attractive benefits to the co-browsing participant. One is that the co-browsing participant does not need to have the capability of establishing network connection with the original Web server (the connection marked in step 7 is denoted by a dashed line due to this reason). The other is that if the co-browsing participant has a fast network connection with the co-browsing host (e.g., they are within the same LAN), downloading cached objects from the host browser rather than from the remote Web server can often reduce the response time.

In step 9, any dynamic changes made (e.g., by JavaScript or Ajax) to a co-browsed webpage can be synchronized in real time from the host browser to the participant browser. Meanwhile, one user’s (either a host user or a participant user) browsing actions such as form filling and mouse-pointer moving can be monitored and instantly mirrored to other users. When the co-browsing host visits new webpages, the loop from steps 3 to 9 is repeated. In a co-browsing session, users can visit different websites and collaboratively browse and operate on as many webpages as they like.

3.2 Decisions

The design of the RCB framework is mainly based on three decisions with respect to the communication model, the service model, and the synchronization model, respectively.

3.2.1 Direct Communication Model

Our RCB framework uses a direct communication model to support the collaboration between a co-browsing host and a co-browsing participant. A participant browser establishes a TCP connection to a host browser, without the support of any third-party platform, server, or proxy.

This direct communication model is simple, convenient, and widely applicable. Users in the same LAN can use Ethernet or Wi-Fi to establish their TCP connections. For WAN environments, if the host browser is running on a machine with a resolvable hostname or reachable IP address, remote co-browsing participants can use the hostname or IP address and an allowed TCP port to establish the connections; otherwise, a co-browsing host can still

allow remote participants to reach a TCP port on a private IP address inside a LAN using port-forwarding [29] techniques. We also consider to integrate some NAT (network address translation) traversal techniques into RCB-Agent to further improve its accessibility.

3.2.2 HTTP-based Service Model

In our RCB framework, RCB-Agent on a host browser uses an HTTP-based service model to serve co-browsing participants. The key benefit of using this model is that there is no need for a co-browsing participant to make any installation or configuration. With the direct communication model, other service models (e.g., a peer-to-peer model or a non-HTTP based service model) exist but they all require changes at the participant side.

Integrating this HTTP-based service model into a browser also simplifies the host side installation since a co-browsing host only needs to install an RCB-Agent browser extension. Meanwhile, this browser integration approach maximizes the co-browsing capability because a browser extension normally can access both the content and related events of the browsed webpages. Furthermore, the end-user extensibility provided by modern Web browsers such as Internet Explorer and Firefox makes the implementation of this service model feasible.

3.2.3 Poll-based Synchronization Model

After the connection between a co-browsing host and its participant is established, Ajax-Snippet will periodically poll RCB-Agent to synchronize the co-browsing session. HTTP is a stateless protocol [8], and the communication is initiated by a client. Since the HTTP protocol does not support the push-based synchronization model, we use poll-based synchronization to emulate the effect of pushing webpage content and user interaction information between co-browsing users. In addition to poll-based synchronization, an HTTP server can use “multipart/x-mixed-replace” type of responses to emulate the content pushing effect. However, compared with poll-based synchronization, this alternative approach increases the complexity of co-browsing synchronization and decreases its reliability.

Ajax-Snippet is written in pure JavaScript. All popular Web browsers support Ajax techniques [20] and currently about 95% of Web users turn on JavaScript in their browsers [21]. Therefore, this synchronization model is well supported on users’ regular browsers.

3.3 Co-browsing Topologies and Policies

The use of RCB is very flexible. Each co-browsing host can support multiple participants, and a participant can

join or leave a session at any time. A user can even host a co-browsing session and meanwhile join sessions hosted by other users using different browser windows or tabs. RCB-Agent knows exactly which participants are connected, and it can notify this information to a co-browsing host or participant.

Each co-browsing session is hosted and moderated by a co-browsing host. A participant’s actions such as mouse click and data input are synchronized to the co-browsing host, and the co-browsing host will decide on further navigating actions. A participant browser never leaves the URL address of RCB-Agent, and contents from different websites and webpages are simply pushed to the participant browser. This tightly coupled scenario is typical for co-browsing applications (e.g., online training and customer support) that need a user to preside over a session, and it is also typical for co-browsing applications (e.g., online shopping) that require users to accomplish a common task on session-protected webpages.

To coordinate co-browsing actions among users, RCB-Agent can enforce different high-level policies for different application scenarios. For example, when a participant clicks a link on a co-browsed webpage and this action information is sent back to the host browser, RCB-Agent can either immediately perform the click action on the host browser, or ask the co-browsing host to inspect and explicitly confirm this click action. Similarly, if multiple participants are involved in a co-browsing session, it is up to the high-level policy enforced on RCB-Agent to decide whom are allowed to perform certain interactions and whose interaction action will be finally submitted to a website. However, the specification and enforcement of co-browsing policies is usually application-dependent, and it is out of the scope of this paper.

3.4 Security Design and Analysis

For a co-browsing participant, using RCB is as secure as visiting a trusted HTTP website. This is because a participant only needs to type in the URL address of RCB-Agent given by a trusted co-browsing host and then perform regular browsing actions such as clicking and form-filling on a regular Web browser. We therefore keep the focus of our security design on the protection of RCB-Agent by authenticating its received requests.

Our current design on request authentication is based on a conventional mechanism of sharing a session secret key and computing the keyed-Hash Message Authentication Code (HMAC). On a host browser, a session-specific one-time secret key is randomly generated and used by RCB-Agent. The co-browsing host shares the secret key with a participant using some out-of-band mechanisms such as telephone calls or instant messages. On a participant browser, the secret key is typed in by a co-browsing

participant via a password field on the *initial HTML page* and then stored and used by Ajax-Snippet.

Before sending a request, Ajax-Snippet computes an HMAC for the request and appends the HMAC as an additional parameter of the request-URI. After receiving a request sent by Ajax-Snippet, RCB-Agent computes a new HMAC for the received request (discarding the HMAC parameter) and verifies the new HMAC against the HMAC embedded in the request-URI. The data integrity and the authenticity of a request are assured if these two HMACs are identical. Since the size of a request sent by Ajax-Snippet is small, an HMAC can be efficiently calculated and any important information in a request can also be efficiently encrypted using a JavaScript implementation [25]. However, using JavaScript to compute an HMAC for a response (or encrypt/decrypt a response) is inefficient, especially if the size of the response is large. We plan to integrate other security mechanisms to address this issue in the future.

4 Implementation Details

Although the design of the proposed RCB framework is relatively simple and straightforward, its implementation poses several challenges. The implementation of RCB-Agent has two major challenges: (1) how to efficiently process requests so that participant browsers can be synchronized in real time, and (2) how to accurately generate response contents so that fine-grained high-quality co-browsing activities can be easily supported. The key implementation challenge of Ajax-Snippet lies in how to properly and smoothly update webpage contents on a participant browser. We have implemented the RCB framework in Firefox and successfully addressed these challenges. We present the implementation details of the framework in this section.

4.1 RCB-Agent

RCB-Agent is implemented as a Firefox browser extension, and it is purely written in JavaScript. Its request processing and response content generation functionalities are detailed as follows.

4.1.1 Request Processing

The request processing functionality of RCB-Agent is implemented as a JavaScript object of Mozilla’s nsIServerSocket interface [33]. This interface provides methods to initialize a server socket and maintain it in the listening state. For this server socket object, we create a socket listener object which implements the methods of Mozilla’s nsIServerSocketListener interface [33]. RCB-Agent uses this socket listener object to asynchronously

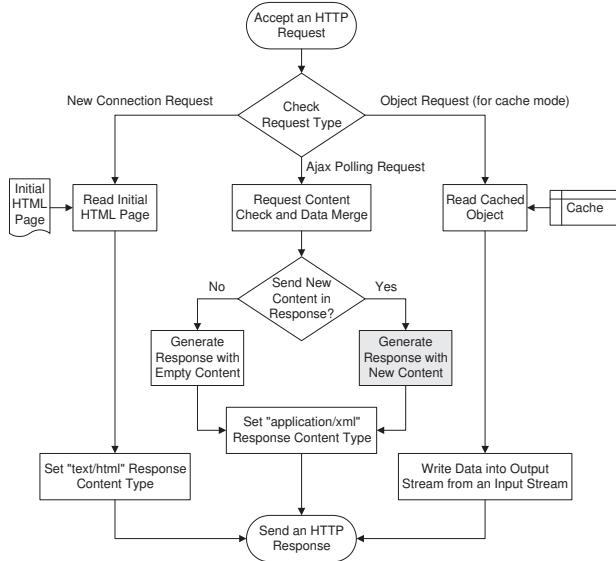


Figure 2: Request processing procedure of RCB-Agent.

listen for and accept new TCP connections. We also create a data listener object which implements Mozilla’s nsIStreamListener interface [33]. We associate this data listener object with the input stream of each connected socket transport. Therefore, over each accepted TCP connection, RCB-Agent uses this data listener object to asynchronously accept incoming HTTP requests and efficiently process them.

Figure 2 illustrates the high-level request processing procedure of RCB-Agent. From a participant browser, RCB-Agent may receive three types of HTTP requests: a *new connection request*, an *object request*, and an *Ajax polling request*. RCB-Agent identifies the type of request by simply checking the *method* token and *request-URI* token in the request-line [8]. Both a new connection request and an object request use the “GET” method, but they can be differentiated by checking their request-URI tokens. The former has a root URI, but the later has a URI pointing to a specific resource such as an image file. Ajax polling requests always use the “POST” method because we want to directly piggyback action information of a co-browsing participant onto a polling request.

A new connection request is sent to RCB-Agent after the URL of RCB-Agent is entered into the address bar of a participant browser. RCB-Agent responds to this request by sending back a “text/html” type of HTTP response to the participant browser with the content of an *initial HTML page*. The head element of this initial HTML page contains Ajax-Snippet, which will later send Ajax polling requests to RCB-Agent periodically.

An object request is sent to RCB-Agent if the *cache mode* is used to allow a participant browser to directly download a cached object from the host browser. RCB-

Agent keeps a mapping table, in which the request-URI of each cached object maps to a corresponding cache key. After obtaining the cache key for a request-URI, RCB-Agent reads the data of a cached object by creating a cache session via Mozilla’s cache service [33]. To save time and memory, RCB-Agent directly writes data from the input stream of the cached object into the output stream of the connected socket transport.

An Ajax polling request is sent by Ajax-Snippet from a participant browser to check if any page content changes or browsing actions have occurred on the host browser. RCB-Agent follows three steps to process an Ajax polling request: data merging, timestamp inspection, and response sending.

Data merging: RCB-Agent examines the content of a “POST” type Ajax polling request and may merge data if the content contains piggybacked browsing action information of the co-browsing participant. For example, if users are co-filling a form, the form data submitted by a co-browsing participant can be extracted and merged into the corresponding form on the host browser.

Timestamp inspection: RCB-Agent looks for any new content needs to be sent back to the co-browsing participant. RCB-Agent uses a simple timestamp mechanism to ensure that only new content, which has never been sent to this participant before, is included in the response message. A timestamp used here is the number of milliseconds since midnight of January 1, 1970. RCB-Agent maintains a timestamp for the latest webpage content on the host browser. Whenever this new content is sent to a participant browser, its timestamp is also included in the same response. Each Ajax polling request from a participant browser carries back the timestamp of its current webpage content, so RCB-Agent can compare the current timestamp on the host browser and the received one to accurately determine whether the page content on each particular participant browser needs to be updated.

Response sending: if any new content needs to be sent to a participant browser, RCB-Agent generates a response with the new content. Response content generation is an important functionality of RCB-Agent, and it is detailed in the following subsection. To facilitate efficient content parsing in a participant browser, RCB-Agent sends out the new content in the form of an XML document using the “application/xml” type of HTTP response. If no new content needs to be sent back, RCB-Agent sends a response with empty content to the participant browser in order to avoid hanging requests.

4.1.2 Response Content Generation

The response content generation functionality of RCB-Agent generates responses with new content for Ajax polling requests. It guarantees that webpage content can

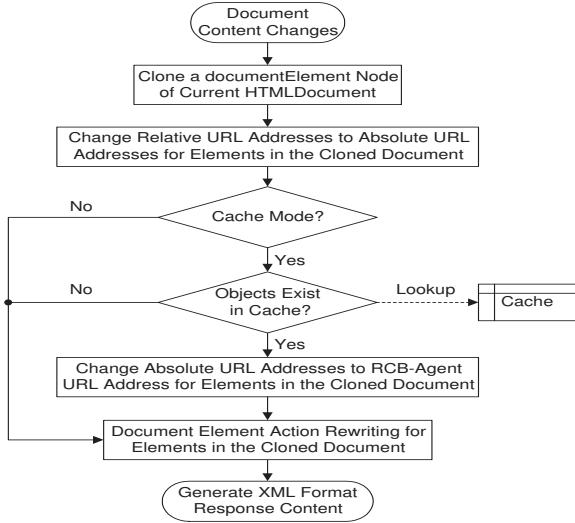


Figure 3: Response content generation procedure of RCB-Agent.

be efficiently extracted on a host browser and later on accurately rendered on a participant browser. The high-quality implementation of this functionality is essential for adding upper-level co-browsing features such as form co-filling and action synchronization.

Figure 3 illustrates the high-level response content generation procedure of RCB-Agent. When document content changes on the host browser need to be sent to a participant browser, RCB-Agent uses the following five steps to generate the XML format response content. First, RCB-Agent clones the *documentElement* node (namely the “*<html>*” root element of an HTML webpage) of the current *HTMLDocument* object on the host browser. The following changes are made only to the cloned *documentElement* node (referred to as the cloned document) so that the content generation procedure will not cause any state change to the current document on the host browser.

In the second step, for the supplementary objects of the cloned document, RCB-Agent changes all the relative URL addresses to absolute URL addresses of the original Web servers. This URL conversion is necessary to support RCB’s *non-cache* mode in which a participant browser needs to use absolute URL addresses to correctly download supplementary objects from original Web servers. To achieve an accurate URL conversion, we create an observer object which implements the methods of Mozilla’s *nsIObserverService* [33]. Using this observer object, RCB-Agent can record complete URL addresses for all the object downloading requests.

In the third step, if the cache mode is used, for the supplementary objects of the cloned document that exist in the browser cache, their absolute URL addresses are changed to RCB-Agent URL addresses. Subsequently,

```

<?xml version='1.0' encoding='utf-8'?>
<newContent>
<docTime>documentTimestamp</docTime>
<docContent>
<docHead>
<hChild1><! [CDATA[escape (hData1) ]]></hChild1>
<hChild2><! [CDATA[escape (hData2) ]]></hChild2>
.....
</docHead>
<!-- for a page using body element -->
<docBody><! [CDATA[escape (bData) ]]></docBody>
<!-- for a page using frames -->
<docFrameSet><! [CDATA[escape (fData) ]]>
</docFrameSet>
<docNoFrames><! [CDATA[escape (nData) ]]>
</docNoFrames>
</docContent>
<userActions>userActionData</userActions>
<newContent>
  
```

Figure 4: XML format response content.

when a participant browser renders the page content, it will automatically send “GET” type of HTTP requests to RCB-Agent to retrieve cached objects. For the non-cache mode, nothing needs to be done in this step. Switching between these two modes is very flexible and fully controlled by RCB-Agent. For example, RCB-Agent can allow different participant browsers to use different modes, allow different webpages sent to a particular participant browser to use different modes, and even allow different objects on the same webpage to use different modes.

In the fourth step, RCB-Agent rewrites event attributes such as *onclick* and *onsubmit* for children elements of the cloned document. The purpose of this rewriting is to enable upper-level co-browsing features such as form co-filling and action synchronization. For instance, to support the form co-filling feature, RCB-Agent changes the *onsubmit* event attribute values of form elements in the cloned document. More specifically, RCB-Agent adds a call to a specific JavaScript function residing in Ajax-Snippet to each form’s *onsubmit* event handler. So later on, when a form is submitted on a participant browser, this JavaScript function is called and the related form data can be carried back by an Ajax polling request to the host browser.

Finally, after making the above changes, RCB-Agent generates an XML format response content for this Ajax polling request. From top-level children of the cloned document, RCB-Agent follows their order in the DOM (Document Object Model [23]) tree to process these elements, including extracting their attribute name-value lists and *innerHTML* values. For most webpages, the cloned document only contains two top-level children: a head element and a body element. For some webpages, their top-level children may include a head element, a frameset element, and probably a noframes element.

Figure 4 illustrates the simplified XML format of the generated response content. The *newContent* element contains a *docTime* element that carries the document

timestamp string, a *docContent* element that carries the data extracted from the cloned document, and a *userActions* element that can carry additional browsing action (such as mouse-pointer movement) information.

Within the *docContent* element, for each child element of the cloned document head, its attribute name-value list and innerHTML value are encoded using the JavaScript *escape* function and carried inside the *CDATA* section of a corresponding *hChild* element. For example, *hChild1* may contain the data for the title child element of the head, and *hChild2* may contain the data for a style element of the head. The contents of these children head elements are separately transmitted so that later Ajax-Snippet can properly and easily update document contents on different types of browsers such as Firefox and Internet Explorer. Similarly, the name-value lists and innerHTML values extracted from other top-level children (e.g., body or frameset) of the cloned document are carried in the *CDATA* sections of their respective elements. We use the *escape* encoding function and *CDATA* section to ensure that the response data can be precisely contained in an “application/xml” message and correctly transmitted over the Internet.

The generation of this XML format response content combines both the structural advantages of using DOM and the performance and simplicity advantages of using innerHTML. This implementation ensures that the response content can be efficiently generated on a host browser; more importantly, it guarantees the same webpage content can be accurately and efficiently rendered on a participant browser. The innerHTML property is well supported by all popular browsers and has been included into the HTML 5 DOM specification. Note that the whole response content generation procedure is executed only once for each new document content, and the generated XML format response content is reusable for multiple participant browsers. Also note that RCB-Agent does not replicate HTTP cookies or the *referer* request header to a participant browser. We can extend RCB-Agent to have these capabilities, but in our experiments we did not observe the necessity to do so because a participant browser can download supplementary objects of a webpage from a website (in the non-cache mode) or RCB-Agent (in the cache mode) for both HTTP and HTTPS sessions.

4.2 Ajax-Snippet

Ajax-Snippet is implemented as a set of JavaScript functions. It is embedded in the head element of the initial HTML page and sent to a participant browser as a part of RCB-Agent’s response to a new connection request. Ajax-Snippet uses the XMLHttpRequest object [31] to asynchronously exchange data with RCB-Agent.

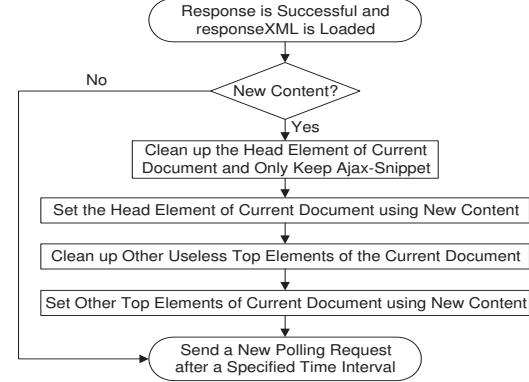


Figure 5: Response processing procedure of Ajax-Snippet.

4.2.1 Ajax Request Sending

Sending Ajax requests is relatively simple for Ajax-Snippet. The first Ajax request is sent after the initial HTML page is loaded on a participant browser. Each following Ajax request is triggered after the response to the previous one is received. A new XMLHttpRequest object is created to send each Ajax request. An “onreadystatechange” event handler is registered for an XMLHttpRequest object to asynchronously process its *readystatechange* events. The XMLHttpRequest object uses the “POST” method so that action information of a co-browsing participant can be directly piggybacked in an Ajax polling request. Before a request is sent out, its *Content-Length* request header needs to be correctly set.

4.2.2 Ajax Response Processing

It is more challenging for Ajax-Snippet to properly process Ajax responses and smoothly update webpage content on a participant browser. Figure 5 illustrates the high-level response processing procedure of Ajax-Snippet. This procedure is implemented in the “onreadystatechange” event handler. It is triggered when a response is successful (HTTP status code sent by RCB-Agent is 200) and the data transfer has been completed (*readyState* is “DONE” and *responseXML* is loaded) for an XMLHttpRequest. If RCB-Agent indicates “no new content” with an empty response content, Ajax-Snippet simply uses the JavaScript *setTimeout* function to send a new polling request after a specified time interval; otherwise, Ajax-Snippet will update the current webpage document in which it resides, using the new content contained in the *responseXML* document.

A new content could be either a brand new webpage or an update to the existing webpage. To make the content update process smooth and simple on a participant browser, Ajax-Snippet follows a specific four-step procedure. First, Ajax-Snippet cleans up other content

in the head element of the current document, but it always keeps itself as a “`<script>`” child element within the head element of any current document. Next, Ajax-Snippet extracts the attribute name-value lists and inner-HTML values from the `docHead` element of the new content (shown in Figure 4) and appends them to the head element of the current document. Ajax-Snippet detects browser capability and executes this step differently to best accommodate different browser types. For example, since the `innerHTML` property of the head element is writable in Firefox, Ajax-Snippet will directly set the new value for it. In contrast, the `innerHTML` property is read-only for the head element (and its style child element) in Internet Explore, so Ajax-Snippet will construct each child element of the head element using DOM methods (e.g., `createElement` and `appendChild`).

After properly updating the content of the head element in the above two steps, Ajax-Snippet will then check the new content and clean up other useless top-level elements of the current document. For example, if the current document uses a body top-level element while the new content contains a new webpage with a frameset top-level element, Ajax-Snippet will remove the body node of the current document. Finally, Ajax-Snippet sets other attribute name-value lists and inner-HTML values of the current document based on the data extracted from the new content, following their order in the XML format.

The above procedure ensures that the webpage content on a participant browser can be accurately and smoothly synchronized to that on the host browser. Meanwhile, Ajax-Snippet always resides in the current webpage on a participant browser to maintain the communication with the host browser. After updating the current document with the new content, Ajax-Snippet sends a new polling request to RCB-Agent, in the same way as it does for the “no new content” case.

It is also worth mentioning that any dynamic DOM changes on a host browser are synchronized to a participant browser. Since Ajax-Snippet updates the content mainly using `innerHTML`, the code between a pair of “`<script>`” and “`</script>`” tags will not be executed automatically in both Firefox and Internet Explore. However, event handlers previously rewritten by RCB-Agent can be triggered. The executions of these event handlers on a participant browser will not directly update any URL or change the DOM; they just ask Ajax-Snippet to send action information back to the host browser.

5 Evaluations

In this section, we present the performance evaluation and usability study of our RCB framework.

5.1 Performance Evaluation

To quantify the performance of our RCB framework, we conducted two sets of experiments: one in a LAN environment and the other in a WAN environment.

5.1.1 Experimental Methods

The homepages of 20 sample websites (shown in Table 1) were used for co-browsing experiments. These websites were chosen from the top 50 sites listed by Alexa.com, with a few diversity-related criteria (such as geographical location and content category) taken into consideration.

We introduce six metrics to evaluate the real-time performance of the RCB framework: **M1**, the time used by a host browser to load the HTML document of a homepage from a Web server; **M2**, the time used by a participant browser to load the content of the same HTML document from the host browser; **M3**, the time used by the participant browser to download the supplementary Web objects (of the HTML document) in the non-cache mode; **M4**, the time used by the participant browser to download the supplementary Web objects (of the HTML document) in the cache mode; **M5**, the time used by the host browser to generate the response content for an HTML document; **M6**, the time used by the participant browser to update its current document based on the new content of an HTML document.

Intuitively, the metric M1 measures the download speed of an HTML document while the metric M2 measures the synchronization speed of the HTML document. We use M3 and M4 to determine whether using the cache mode is beneficial to a co-browsing participant. The metrics M5 and M6 quantify the speed of RCB-Agent in response content generation (i.e., the procedure illustrated in Figure 3) and the speed of Ajax-Snippet in response processing (i.e., the procedure illustrated in Figure 5), respectively. User browsing action information (such as form co-filling data) can be carried in a small-sized request or response and efficiently transmitted, so we do not present the detailed results.

In each experimental environment, we used one host browser and one participant browser. The polling time interval of Ajax-Snippet was set to one second, which we believe is small enough because users’ average think time on a webpage is about ten seconds [16]. We co-browsed all the 20 sample sites in the cache mode for the first round and then in the non-cache mode for the second round. Both browsers were directly connected to the Internet without using any proxies. Before each round of co-browsing, the caches of both browsers were cleaned up. This procedure was repeated five times and we present the average results.

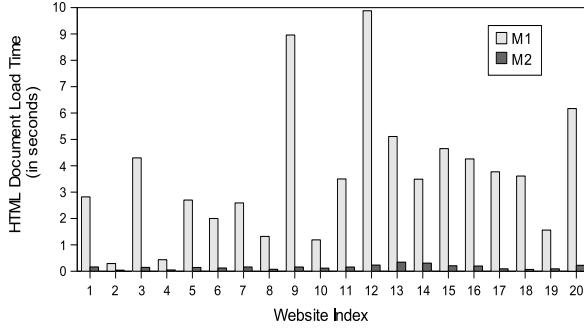


Figure 6: HTML document load time in the LAN environment.

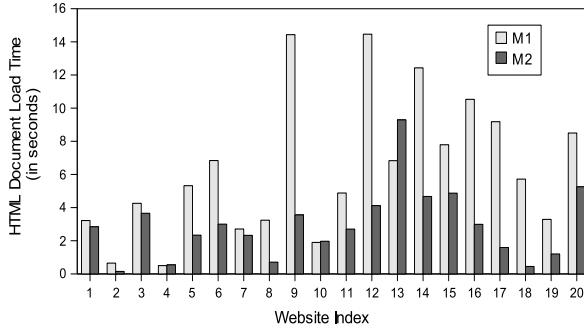


Figure 7: HTML document load time in the WAN environment.

5.1.2 Experimental Results

The first set of experiments were conducted in a 100Mbps Ethernet LAN environment, where the host and participant PCs resided in the same campus network. The second set of experiments were performed in a WAN environment, where the host and participant PCs resided in two geographically separated homes. Both homes used slow speed Internet access services with 1.5Mbps download speed and 384Kbps upload speed.

Figure 6 shows the comparison between metrics M1 and M2 in the LAN environment, and Figure 7 presents the same comparison in the WAN environment. In the LAN environment, for all the 20 sample sites, the values of M2 are less than 0.4 seconds, which are much smaller than those of M1. In other words, the HTML document content synchronization delay experienced by the participant browser is much smaller than the time it has to spend to directly download the HTML document from a remote Web server. This result is expected since the host PC and participant PC were in the same LAN. In the WAN environment, the values of M2 become larger than those in the LAN environment. This is mainly because the upload link speed at the host PC side was slow (only 384Kbps). However, we can see that most values of M2 (17 out of 20 sample sites) are still smaller than those of M1, indicating an acceptable content synchronization speed.

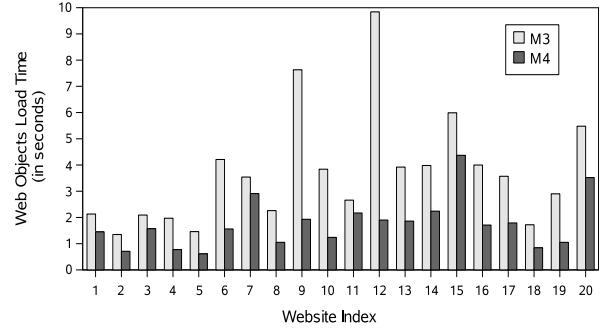


Figure 8: Cache mode performance gain in the LAN environment.

Index	Site Name	Page Size (KB)	M5 non-cache (second)	M5 cache (second)	M6 (second)
1	yahoo.com	130.3	0.066	0.098	0.135
2	google.com	6.8	0.015	0.020	0.045
3	youtube.com	69.2	0.107	0.172	0.126
4	live.com	20.9	0.019	0.037	0.057
5	msn.com	49.6	0.079	0.145	0.119
6	myspace.com	53.2	0.085	0.097	0.126
7	wikipedia.org	51.7	0.113	0.138	0.171
8	facebook.com	23.2	0.029	0.036	0.067
9	yahoo.co.jp	101.4	0.111	0.156	0.154
10	ebay.com	50.5	0.049	0.098	0.100
11	aol.com	71.3	0.099	0.189	0.142
12	mail.ru	83.8	0.176	0.346	0.268
13	amazon.com	228.5	0.371	0.687	0.318
14	cnn.com	109.4	0.298	0.599	0.280
15	espn.go.com	110.9	0.175	0.376	0.194
16	free.fr	70.0	0.211	0.279	0.222
17	adobe.com	37.3	0.050	0.085	0.086
18	apple.com	10.0	0.029	0.056	0.118
19	about.com	35.8	0.056	0.100	0.081
20	nytimes.com	120.0	0.221	0.382	0.196

Table 1: Homepage size and processing time of 20 sites.

Figure 8 illustrates the comparison between metrics M3 and M4 in the LAN environment. We can see that the values of M4 are less than those of M3 for all the 20 sample sites. It means that for the participant browser, downloading the supplementary Web objects from the host browser is faster than retrieving them from the remote Web server. This result is expected as well since the co-browsing PCs were in the same LAN. Therefore, we suggest to turn on the cache mode in LAN environments so that co-browsing participants can take advantage of the performance gain provided by cache. In the WAN environment, co-browsing participants can still benefit from the cache at the host side although the performance gain is not as significant as that in the LAN environment. We omit the details to save space.

Table 1 lists the homepage size of the sample sites and the processing time in terms of the M5 metric for both the non-cache mode and cache mode, and the M6 metric. Based on the results in the table, we have the following observations. First, the larger the HTML document size is, the more processing time is needed. Second, RCB-Agent can efficiently generate the response content for an HTML document. Most pages (16 out of 20 for M5

non-cache, and 14 out of 20 for M5 cache) can be processed in less than 0.2 seconds. Since a generated new content can be reused by multiple co-browsing participants, this processing time on the host browser is reasonably small. Third, RCB-Agent needs more processing time in the cache mode than in the non-cache mode, i.e., the values of M5 cache are greater than those of M5 non-cache. This is because extra cache lookup time is spent in the cache mode. However, this small cost is outweighed by the benefits of using the cache-mode for co-browsing participants, especially in LAN environments as shown above. Finally, Ajax-Snippet can efficiently update webpage content on a participant browser. As indicated by the values of the M6 metric, this processing time is less than one-third of a second for all the 20 webpages.

5.2 Usability Evaluation

To measure whether our RCB framework is helpful and easy to use, we conducted a usability study based on two real co-browsing scenarios: (1) coordinating a meeting spot via Google Maps, and (2) online co-shopping at Amazon.com. In the remainder of this section, we first introduce these two scenarios and explain why we chose them. We then present and analyze the usability study.

5.2.1 Coordinating a Meeting Spot via Google Maps

Suppose Alice is going to visit New York City. She plans to meet her local cousin Bob at the Cartier jewelry store on the Fifth Avenue in Manhattan to buy a watch. Bob wants to use Google Maps to show Alice the direction to the store. Since the neighborhood around the Fifth Avenue in Manhattan is extremely crowded, Bob uses our RCB tool to give Alice accurate directions to the exact meeting spot.

Bob hosts a co-browsing session and Alice joins the session. Bob then searches the store address using Google Maps. He may zoom in and out of the map, drag the map, and show different views of the map. Whatever content Bob is seeing on his browser is instantly and accurately synchronized to Alice's browser. Figure 9 shows one snapshot of the destination map shown on Alice's browser. Bob may even use the street-view Flash of Google Maps to show Alice panoramic street-level views of the meeting spot. Note that our current implementation does not support the synchronization of users' actions on a Flash, so Alice and Bob can only individually operate on a Flash. During the session, they may use an instant message tool (e.g., MSN Messenger) or telephone as the supplementary communication channel to mediate actions. Eventually Alice and Bob come to the agreement that they will meet outside the four red roof show-windows of Cartier on the Fifth Avenue side.

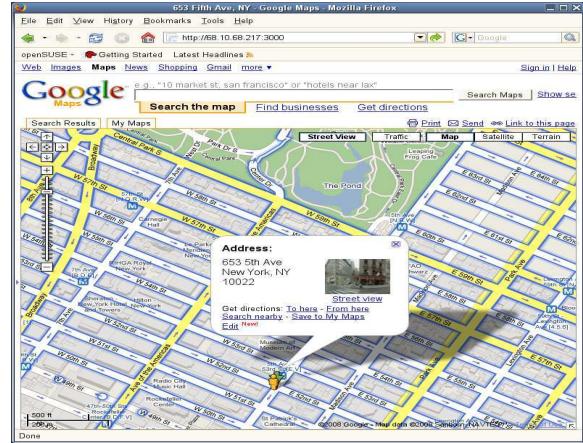


Figure 9: Snapshot of the destination map shown on Alice's browser.

This scenario exemplifies that our RCB framework can efficiently support rich Web contents and communication intensive webpages. Google Maps actually also uses Ajax to asynchronously retrieve small images (usually in the size of 256 by 256 pixels) and smoothly update the map content grid by grid. With our RCB tool, one user's view is further synchronized accurately and smoothly to another user's browser, achieving real-time collaborative browsing. In general, the URL in the address bar remains the same even if the webpage content has been updated by Ajax and many other DHTML (Dynamic HTML) techniques. Therefore, without RCB, the map content changes caused by Bob's browsing actions such as zooming and panning cannot be synchronized to Alice by simply sharing URLs.

5.2.2 Online Co-shopping at Amazon.com

Bob is going to buy a present for his cousin Alice. Bob hosts a co-browsing session and Alice joins the session. They co-browse a number of webpages at Amazon.com to select a newly-released MacBook Air laptop favored by Alice. Both Alice and Bob can type in, search and click on a webpage. Bob's browsing requests will be directly sent to Amazon.com, but Alice's action information such as searching or clicking is first sent back to the RCB-Agent on Bob's browser and then sent out to Amazon.com. After they made the decision, Bob adds the selected laptop to the shopping cart and uses his account to start the checkout procedure. Bob can ask Alice to co-fill some forms (e.g., the shipping address form) using her information, and he finishes the rest of the checkout procedure. Figure 10 shows the snapshot of the form filling window on Bob's browser, on which the form data is sent back from Alice's browser.

The online shopping scenario verifies that our RCB tool can: (1) correctly synchronize webpages with very

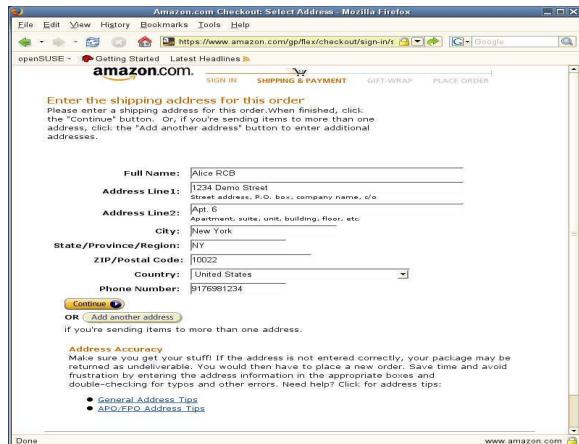


Figure 10: Snapshot of the form filling window on Bob’s browser.

complicated layout and dynamically-generated content, (2) allow anyone in a co-browsing session to initiate browsing actions and navigate to new pages, (3) support co-browsing features such as form co-filling and mouse clicking, and (4) support session-protected webpages.

5.2.3 Usability Study

The main objective of the usability study is to measure whether our RCB tool is helpful and easy to use.

(1) Test subjects: A total of 20 adults, 11 females and 9 males, participated as users in our study. These test subjects were undergraduate and graduate students who were randomly recruited from nine degree programs at our university. Eighteen test subjects were between ages of 18 and 30, and two were over 30 years old. Nineteen test subjects were using the Internet daily, and one was using it weekly. We did not screen test subjects based on experience using Firefox because they simply had to perform tasks (such as entering URLs and interacting with webpages) that are common to different browsers. We also did not screen test subjects based on experience using Google Maps or shopping at Amazon.com.

(2) Procedure and Tasks: We combined the two scenarios (Google Maps and Amazon.com) introduced above into a single co-browsing session. Each session consists of 20 tasks as listed in Table 2. Ten tasks were performed by Bob and ten tasks were performed by Alice, and Alice and Bob represent two role-players regardless of their actual genders. The 20 test subjects were randomly grouped into 10 pairs. We asked each pair of test subjects to complete two sessions. In the first session, we randomly asked one test subject to act as Alice and the other test subject to act as Bob. After the two test subjects finished the 20 tasks in a session, they switched their roles to perform the 20 tasks in the second session.

The two test subjects in a pair were asked to use two computers located at different locations either in our de-

Task#	Brief Task Description
T1-B	Bob starts a RCB co-browsing session using a Firefox browser.
T1-A	Alice types the URL told by Bob in a Firefox browser to join the session.
T2-B	Bob searches the location “653 5th Ave, New York” using Google Maps.
T2-A	Alice tells Bob that the map of the location is automatically shown on her browser.
T3-B	Bob zooms in and out of the map, drags up/down/left/right the map.
T3-A	Alice tells Bob that the map is automatically updated on her browser.
T4-B	Bob clicks to the street-view of the searched location.
T4-A	Alice tells Bob that the street-view is also automatically shown on her browser.
T5-B	Bob tells Alice to meet outside the four red roof show-windows of Cartier shown in the street-view.
T5-A	Alice finds the four red roof show-windows of Cartier and agrees with the meeting spot.
T6-B	Bob continues to visit the homepage of Amazon.com website.
T6-A	Alice tells Bob that the homepage of Amazon.com is automatically shown on her browser.
T7-B	Bob searches and clicks to find a MacBook Air laptop at the Amazon.com website.
T7-A	Alice tells Bob that the pages are automatically updated on her browser.
T8-B	Bob asks Alice to search and click on the pages shown on her browser to choose a different MacBook Air laptop.
T8-A	Alice chooses a different MacBook Air laptop and tells Bob that this laptop is her final choice.
T9-B	Bob adds the selected laptop to the shopping cart and starts the checkout procedure.
T9-A	Alice fills the shipping address form shown on her browser.
T10-B	Bob finishes the rest of the checkout procedure.
T10-A	Alice leaves the co-browsing session.

Table 2: The 20 tasks used in a co-browsing session. Alice and Bob are two role-players. Bob performs ten tasks from T1-B to T10-B, and Alice performs ten tasks from T1-A to T10-A. Bob and Alice use a voice supplementary communication channel to mediate actions.

partment or in the library of university. We pre-installed RCB-Agent to the Firefox browser on Bob’s computer so that we can keep the focus of the study on using the RCB tool itself. Before a pair of test subjects started performing the tasks, we explained the main functionality of RCB and how to use it. We also gave them an instruction sheet that describes the two scenarios and lists the tasks to be completed by a role-player.

(3) Data Collection: We collected data in two ways: through observation and through two questionnaires. During each co-browsing session, two experimenters sat with each test subject to observe the progress of the tasks. After completing two co-browsing sessions, each test subject was asked to answer a five-point Likert-scale (Strongly disagree, Disagree, Neither agree nor disagree, Agree, Strongly Agree) [27] questionnaire. The 16 questions in this questionnaire are listed in Table 3. In addition to this close-ended questionnaire, each test subject was also asked to answer an open-ended questionnaire to solicit additional feedback. After finishing the two questionnaires and before leaving, each test subject was given a \$5 gift card as compensation for the participation.

(4) Results and Analysis: Through observation, we found that the 10 pairs of test subjects successfully completed all their co-browsing sessions. Each pair of test subjects took an average of 10.8 minutes to complete

Perceived Usefulness
Q1-P: It is helpful to use RCB to coordinate a meeting spot via Google Maps.
Q1-N: It is useless to use RCB to coordinate a meeting spot via Google Maps.
Q2-P: It is helpful to use RCB to perform online co-shopping at Amazon.com.
Q2-N: It is useless to use RCB to perform online co-shopping at Amazon.com.
Ease-of-use as a co-browsing host
Q3-P: It is easy to use RCB to host the Google Maps scenario.
Q3-N: It is hard to use RCB to host the Google Maps scenario.
Q4-P: It is easy to use RCB to host the online co-shopping scenario.
Q4-N: It is hard to use RCB to host the online co-shopping scenario.
Ease-of-use as a co-browsing participant
Q5-P: It is easy to participate in the RCB Google Maps scenario.
Q5-N: It is hard to participate in the RCB Google Maps scenario.
Q6-P: It is easy to participate in the RCB online co-shopping scenario.
Q6-N: It is hard to participate in the RCB online co-shopping scenario.
Potential Usage
Q7-P: It would be helpful to use RCB on other co-browsing activities.
Q7-N: It wouldn't be helpful to use RCB on other co-browsing activities.
Q8-P: I would like to use RCB in the future.
Q8-N: I wouldn't like to use RCB in the future.

Table 3: The 16 close-ended questions in four groups. Test subjects were not aware of the groupings. From Q1-P to Q8-P are eight positive Likert questions, and from Q1-N to Q8-N are eight correspondingly inverted negative Likert questions. These questions were presented to a test subject in random order to reduce response bias.

two sessions. Such a 100% success ratio may be attributable to two main reasons. One is that all the 20 test subjects were frequent Internet users and they might be familiar with online shopping and Web mapping service sites. The other reason is that RCB does not add any new user interface artifact and users simply use regular Web browsers, visit regular websites, and perform regular browsing activities.

A summary of the responses to the 16 close-ended questions is presented in Table 4. Since the data collected are ordinal and do not necessarily have interval scales, we used the median and mode to summarize the data and used the percentages of responses to express the variability of the results. Overall, the test subjects were very enthusiastic about this RCB tool. The median and mode responses are positive *Agree* for all the questions. In terms of the perceived usefulness (Q1-P, Q1-N, Q2-P, Q2-N), 52.5% of responses agree and 40.0% of responses strongly agree that it is helpful to use RCB in both the Google Maps scenario and the Amazon.com scenario.

In terms of the ease-of-use as a co-browsing host (Q3-P, Q3-N, Q4-P, Q4-N), 50.0% of responses agree and 40.0% of responses strongly agree that it is easy to use RCB to host the Google Maps scenario, and 62.5% of responses agree and 27.5% of responses strongly agree that it is easy to use RCB to host the online co-shopping scenario. In terms of the ease-of-use as a co-browsing participant (Q5-P, Q5-N, Q6-P, Q6-N), 62.5% of responses agree and 35.0% of responses strongly agree that it is easy to participate in the RCB Google Maps scenario, and 57.5% of responses agree and 35.0% of responses

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly Agree	Median	Mode
Q1-P	0.0%	0.0%	7.5%	52.5%	40.0%	Agree	Agree
Q2-P	0.0%	0.0%	7.5%	52.5%	40.0%	Agree	Agree
Q3-P	5.0%	0.0%	5.0%	50.0%	40.0%	Agree	Agree
Q4-P	0.0%	2.5%	7.5%	62.5%	27.5%	Agree	Agree
Q5-P	0.0%	2.5%	0.0%	62.5%	35.0%	Agree	Agree
Q6-P	0.0%	5.0%	2.5%	57.5%	35.0%	Agree	Agree
Q7-P	0.0%	2.5%	5.0%	55.0%	37.5%	Agree	Agree
Q8-P	0.0%	0.0%	15.0%	55.0%	30.0%	Agree	Agree

Table 4: Summary of the responses to the 16 close-ended questions. To provide statistical coherence, we inverted the scores to the eight negative Likert questions (Q1-N to Q8-N) about the neutral mark (i.e., Strongly agree to Strongly disagree, Agree to Disagree, and vice versa) and then merged them with the scores to the corresponding positive Likert questions (Q1-P to Q8-P).

strongly agree that it is easy to participate in the RCB online co-shopping scenario. These two groups of results also indicate that participating a co-browsing session is slightly easier than hosting a session.

In terms of the potential usage (Q7-P, Q7-N, Q8-P, Q8-N), 55.0% of responses agree and 37.5% of responses strongly agree that it would be helpful to use RCB on other co-browsing activities, and 55.0% of responses agree and 30.0% of responses strongly agree that the test subject would like to use RCB in the future.

In our open-ended questionnaire, the test subjects were asked to write down whatever they think about the RCB tool. One test subject did not write anything, but nineteen test subjects wrote many positive comments such as “cool”, “it helps cooperation”, “useful”, “simple operation”, and “love it, fascinating and useful”. Meanwhile, some test subjects also wrote a few suggestions and expectations to the RCB tool. For example, two test subjects suggested that indicators of the other person’s connection and status may be needed. Four test subjects mentioned that it would be great if actions in the Google Maps street-view Flash could also be synchronized. Seven test subjects expressed that on some pages the wait time is a bit long, but it is not bad at all.

In summary, the results of the usability study clearly demonstrate that RCB is very helpful and easy to use. It is a simple and practical real-time collaborative browsing tool that people would like to use in their everyday browsing activities.

6 Conclusion

We have presented a simple and practical framework for Real-time Collaborative Browsing (RCB). Leveraging the power of Ajax techniques and the end-user extensibility of modern Web browsers, RCB enables real-time collaboration among Web users without the involvement of any third-party platforms, servers, or proxies.

A co-browsing host only needs to install an RCB-Agent browser extension, and co-browsing participants just use their regular JavaScript-enabled Web browsers. We detailed the design and the Firefox version implementation of the RCB framework. We measured the real-time performance of RCB through extensive experiments, and we validated its high quality co-browsing capabilities using a formal usability study. The evaluation results demonstrate that our RCB framework is simple, practical, helpful and easy to use.

In our future work, we plan to explore co-browsing in mobile computing environments. We have recently ported our RCB-Agent implementation to the Fennec Web browser, which is the mobile version of Firefox. Our preliminary experiments on a Nokia N810 Internet tablet show that RCB-Agent can also efficiently support co-browsing using mobile devices. Currently we are applying our RCB techniques to enable a few interesting mobile applications. We also plan to implement RCB-Agent on other Web browsers. Enabling direct interactions between Web end-users can create many interesting interactive Internet applications. We believe that further exploring this end-user direct interaction capability and its applications is an important future research direction.

7 Acknowledgments

We thank the anonymous reviewers and our shepherd Niels Provos for their insightful comments and valuable suggestions. We also thank Professor Peter M. Vishston of the Department of Psychology at the College of William and Mary for his generous help in usability study. This work was partially supported by NSF grants CNS-0627339 and CNS-0627340.

References

- [1] ANEIROS, M., AND ESTIVILL-CASTRO, V. Usability of Real-Time Unconstrained WWW-Co-Browsing for Educational Settings. In *Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence* (2005), pp. 105–111.
- [2] APPELT, W. WWW Based Collaboration with the BSCW System. In *Proc. of the 26th Conference on Current Trends in Theory and Practice of Informatics* (1999), pp. 66–78.
- [3] ATTERER, R., SCHMIDT, A., AND WNUK, M. A Proxy-Based Infrastructure for Web Application Sharing and Remote Collaboration on Web Pages. In *Proc. of the IFIP TC13 Interantional Conference on Human-Computer Interaction* (2007), pp. 74–87.
- [4] CABRI, G., LEONARDI, L., AND ZAMBONELLI, F. A proxy-based framework to support synchronous cooperation on the Web. *Softw. Pract. Exper.* 29, 14 (1999), 1241–1263.
- [5] CHANG, M. L. CoBrowse Firefox Add-ons. <https://addons.mozilla.org/en-US/firefox/addon/1469>.
- [6] COLES, A., DELIOT, E., MELAMED, T., AND LANSARD, K. A framework for coordinated multi-modal browsing with multiple clients. In *Proc. of the WWW* (2003), pp. 718–726.
- [7] ESENTHER, A. W. Instant Co-Browsing: Lightweight Real-time Collaborative Web Browsing. In *Proc. of the WWW* (2002), pp. 107–114.
- [8] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MAS-INTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, 1999.
- [9] GRAHAM, T. C. N. GroupScape: Integrating Synchronous Groupware and the World Wide Web. In *Proc. of the IFIP TC13 Interantional Conference on Human-Computer Interaction* (1997), pp. 547–554.
- [10] GRAHAM, T. C. N., URNES, T., AND NEJABI, R. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proc. of the ACM UIST* (1996), pp. 1–10.
- [11] GREENBERG, S., AND ROSEMAN, M. GroupWeb: a WWW browser as real time groupware. In *Proc. of the ACM CHI Companion* (1996), pp. 271–272.
- [12] HAN, R., PERRET, V., AND NAGHSHINEH, M. WebSplitter: a unified XML framework for multi-device collaborative Web browsing. In *Proc. of the ACM CSCW* (2000), pp. 221–230.
- [13] ICHIMURA, S., AND MATSUSHITA, Y. Lightweight Desktop-Sharing System for Web Browsers. In *Proc. of the 3rd International Conference on Information Technology and Applications* (2005), pp. 136–141.
- [14] JACOBS, S., GEBHARDT, M., KETHERS, S., AND RZASA, W. Filling HTML forms simultaneously: CoWeb architecture and functionality. *Comput. Netw. ISDN Syst.* 28, 7-11 (1996), 1385–1395.
- [15] KOBAYASHI, M., SHINOZAKI, M., SAKAIRI, T., TOUMA, M., DAIJAVAD, S., AND WOLF, C. Collaborative customer services using synchronous Web browser sharing. In *Proc. of the ACM CSCW* (1998), pp. 99–109.
- [16] MAH, B. A. An Empirical Model of HTTP Network Traffic. In *Proc. of the INFOCOM* (1997), pp. 592–600.
- [17] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing* 2, 1 (1998), 33–38.
- [18] ROSEMAN, M., AND GREENBERG, S. Building real-time groupware with GroupKit, a groupware toolkit. *ACM Trans. Comput.-Hum. Interact.* 3, 1 (1996), 66–106.
- [19] SAKAIRI, T., SHINOZAKI, M., AND KOBAYASHI, M. CollaborationFramework: A Toolkit for Sharing Existing Single-User Applications without Modification. In *Proc. of the Asian Pacific Computer and Human Interaction* (1998), pp. 183–188.
- [20] Ajax (programming). [http://en.wikipedia.org/wiki/Ajax_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming)).
- [21] Browser Statistics. http://www.w3schools.com/browsers/browsers_stats.asp.
- [22] Cobrowse & Chat for Rich Ajax Applications - Backbase. <http://www.backbase.com/products/ajax-applications/cobrowse>.
- [23] Document Object Model (DOM). <http://www.w3.org/DOM>.
- [24] Firefox Extensions. <http://developer.mozilla.org>.
- [25] <http://point-at-infinity.org>.
- [26] Internet Explorer Browser Extensions. [http://msdn.microsoft.com/en-us/library/aa753587\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa753587(VS.85).aspx).
- [27] Likert scale. http://en.wikipedia.org/wiki/Likert_scale.
- [28] PageShare. <https://www.pageshare.com/web/products/index.html>.
- [29] Port forwarding. http://en.wikipedia.org/wiki/Port_forwarding.
- [30] SamePlace. <http://sameplace.cc/wiki/shared-web-applications>.
- [31] XMLHttpRequest. <http://www.w3.org/TR/XMLHttpRequest>.
- [32] XMPP Standards Foundation. <http://www.xmpp.org>.
- [33] XPCOM. <http://www.xulplanet.com/references/xpcomref>.

The Beauty and the Beast: Vulnerabilities in Red Hat’s Packages

Stephan Neuhaus

Dipartimento di Informatica e Telecomunicazioni
Università degli Studi di Trento
I-38100 Trento, Italy
Stephan.Neuhaus@disi.unitn.it

Thomas Zimmermann

Microsoft Research
One Microsoft Way
Redmond, Washington, USA
tz@acm.org

Abstract

In an empirical study of 3241 Red Hat packages, we show that software vulnerabilities correlate with dependencies between packages. With formal concept analysis and statistical hypothesis testing, we identify dependencies that decrease the risk of vulnerabilities (“beauties”) or increase the risk (“beasts”). Using support vector machines on dependency data, our prediction models successfully and consistently catch about two thirds of vulnerable packages (median recall of 0.65). When our models predict a package as vulnerable, it is correct more than eight times out of ten (median precision of 0.83). Our findings help developers to choose new dependencies wisely and make them aware of risky dependencies.

1 Introduction

The Federal Bureau of Investigation (FBI) estimates that security incidents cost the U.S. industry at least 67 billion dollars every year, according to a joint study [14] with the Computer Security Institute in 2005. While keeping a single program secure is already difficult, software security assurance for a large software distribution is a Herculean task. For example, the Red Hat distribution contains more than three thousand software packages,¹ each potentially vulnerable. The challenge for Red Hat is to stay on top of the flood of patches and new versions. In particular, they need to prioritize work so that available resources can be spent on packages that need attention most urgently; for example, because a critical vulnerability has been fixed and the patched version needs to be distributed.

The efforts by Red Hat are complicated by dependencies between packages. For example, the package *mod_php* needs to be installed before package *mediawiki* can be installed, which is why *mediawiki* depends on *mod_php*. Sometimes dependencies form long chains or are conflicting, which can cause frustration among users, also known as *dependency hell* [5].

In this paper, we show that vulnerabilities correlate with dependencies between software packages. For example, when depending on Python the risk of an application being vulnerable decreases, while the risk increases when depending on PHP or Perl. In addition, we demonstrate how to use dependencies to build prediction models for vulnerabilities. More specifically, our contributions are as follows:

1. *Empirical evidence that vulnerabilities correlate with dependencies.* Our study of 3241 Red Hat packages is the largest study of vulnerabilities ever conducted in terms of number of investigated applications.
2. *Identification of dependencies with positive or negative impact on vulnerabilities.* We combine formal concept analysis with statistical hypothesis testing to find dependencies that increase the chance of a package having vulnerabilities—we call such dependencies “*beasts*”. We also find dependencies that decrease the risk of vulnerabilities—we call such dependencies “*beauties*” (Section 3).
3. *Statistical models to predict vulnerabilities.* We use support vector machines on Red Hat dependency data to predict which packages will have vulnerabilities (classification) and which packages will have the most vulnerabilities (ranking). For classification models, the median precision is 0.83 and the median recall is 0.65. For ranking, the median Spearman correlation is 0.58. These numbers show that the dependencies of a package can indeed predict its vulnerability (Section 4).
4. *Techniques to predict fragile packages.* In early 2008, we predicted that 25 packages will turn vulnerable. In the subsequent six months, vulnerabilities were discovered in 9 out the 25 packages (Section 5).

Understanding how dependencies correlate with vulnerabilities is important to build safe software. When *building new applications*, one can choose which packages are dependable. For example, knowing that Python or Gnome applications have been less prone to vulnerabilities in the past, helps to make the right decisions and to minimize risk early. Even if the dependency is unavoidable, one can plan for the increased risk by allocating more resources for quality assurance.

When *Maintaining existing applications*, being aware of dependencies that likely lead to vulnerabilities helps to prioritize resources. Instead of tracking changes and patches for all packages the application depends on, one only needs to track the risky packages.

In the remainder of this paper, we first describe how the data for our experiments was collected (Section 2). We then provide evidence for the correlation of dependencies with vulnerabilities (Section 3) and show how to build models to predict vulnerable packages (Section 4). Next, we explain how to make predictions more descriptive and how to identify fragile packages, i.e., packages that have not yet had vulnerabilities, but soon will have (Section 5). We continue with some hypotheses on why dependencies may influence vulnerabilities (Section 6) and conclude with related work (Section 7) and a discussion of consequences (Section 8).

2 Data Collection

For the study in this paper, we selected the Red Hat Linux distribution, which consists of several hundred applications bundled in software packages, where each package is in a specific version. Packages are provided in the Red Hat Package Manager (RPM) file format that allows easy download and installation using specific tools. In August 2008, there were 3241 RPMs available from Red Hat.²

To protect its customers, Red Hat issues Red Hat Security Advisories (RHSAs) on the Internet [29]. RHSAs describe *vulnerabilities* that have been found in packages,³ and how to prevent them. A typical RHSA is shown in Figure 1. On the bottom of every RHSA is a list of packages that need to be updated to remove the described vulnerability from the system. We refer to this as a package *having an RHSA*. By iterating over all RHSAs, we collected all packages that were linked to vulnerabilities because they had to be updated as part of an RHSA. We also counted for each package by how many RHSAs it was affected; we use this count as the number of vulnerabilities for a package.

The first RHSA was issued in January 2000. Up until January 2008, there were 1468 RHSAs, which are the primary dataset used for most experiments in this paper (see also Figure 2). The following seven months saw an-

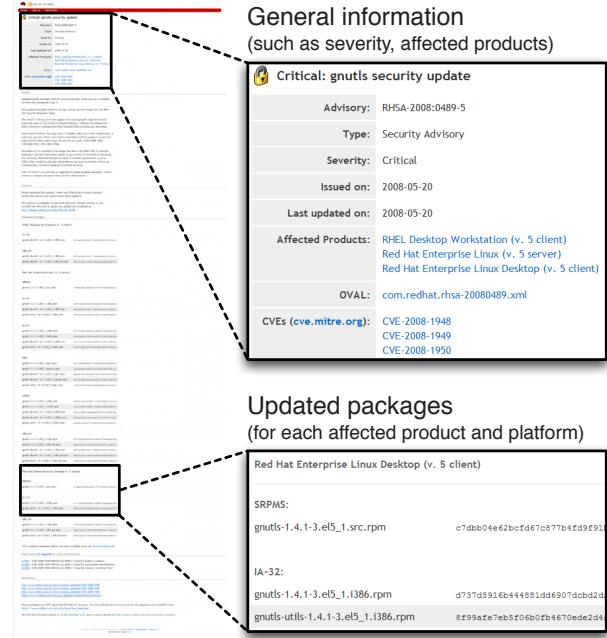


Figure 1: Typical Red Hat Security Advisory.

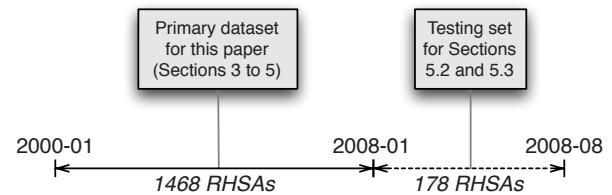


Figure 2: RHSAs used in this paper.

other 178 RHSAs, which we will use as testing set for the prediction of fragile packages (non-vulnerable packages that turn vulnerable) in Sections 5.2 and 5.3. We consider a total of 1646 RHSAs for this paper.⁴

For each package, the RPM file describes which packages are required to be installed. These *dependencies* are stored in so-called tags (type-value pairs) in the RPM header. Each tag with the type `RPMTAG_REQUIRENAME` specifies the name of a dependency.⁵ Extracting dependency information from RPM files is straightforward with the functionality provided by the RPM Library (`rpmlib`) [3]. For our experiments Josh Bressers of the Red Hat Security Response Team generously provided us with a list of dependencies.

We represent the dependency and vulnerability data as follows. If there are n packages in all, dependency data is represented by an $n \times n$ matrix $M = \langle m_{jk} \rangle$, where

$$m_{jk} = \begin{cases} 1 & \text{if package } j \text{ depends on package } k; \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

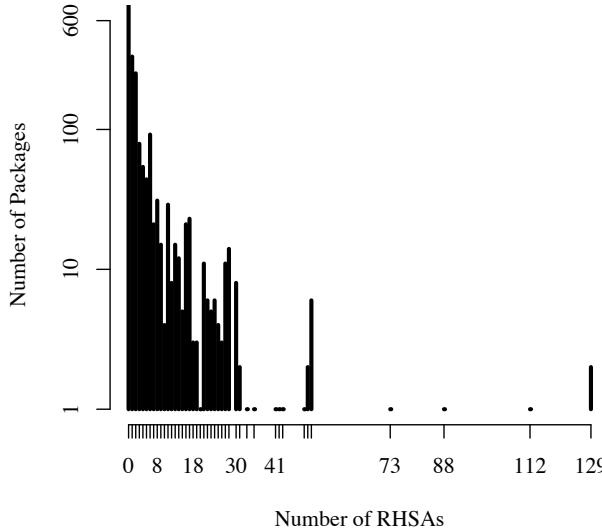


Figure 3: Distribution of RHSAs

The row vector m_j is also called the *dependency vector*. The number of dependencies of a package varies from 0 (typically development packages that contain header files and therefore do not depend on other packages) to a staggering 96 (for *kdebase*), with a median number of 4. The number of dependencies per package looks to be exponentially distributed with a long tail.

Vulnerability information is represented by a vector v with n components where

$$v_k = \text{number of known vulnerabilities in package } k. \quad (2)$$

We call v_j the *associated vulnerability value* for a dependency vector m_j . At the time of writing, there were 1133 packages with and 2108 packages without vulnerabilities. The vulnerable packages were updated a total of 7313 times because of security flaws. The number of updates (RHSAs) per package looks to be exponentially distributed with a long tail (see Figure 3; note the logarithmic y-axis): Many packages needed to be updated only once (332 packages), but 801 packages needed more than one update. The most frequently updated packages were *kernel* and *kernel-doc* with 129 RHSAs. The next most frequently mentioned package was *kernel-smp* with 112 RHSAs. The packages *php-psql*, *php*, *php-ldap*, *php-mysql*, *php-odbc*, and *php-imap* were mentioned in 51 RHSAs.

3 Dependencies and Vulnerabilities

In a first experiment, we applied formal concept analysis (FCA) [10] to the dependency and vulnerability data. FCA takes a matrix as input (in our case M) and returns

all maximum blocks. Each block B consists of two sets of packages O and A . The set O contains the packages that depend on *all packages* in set A , or more formally:⁶

$$\forall o \in O : \forall a \in A : m_{oa} = 1$$

Being a maximum block means, there is no true superset of O for which each package depends on all packages in A and there is no true superset of A on which each package in O depends on.

$$\nexists O' \supset O : \forall o \in O' : \forall a \in A : m_{oa} = 1$$

$$\nexists A' \supset A : \forall o \in O : \forall a \in A' : m_{oa} = 1$$

In total, FCA identifies 32,805 blocks in the Red Hat dataset (for a subset see Figure 4). As an example for a block consider $B_2 = (O_2, A_2)$:

$$O_2 = \{PyQt, ark, arts, avahi-qt3, cervisia, chromium, \dots, 155 \text{ packages in total}\}$$

$$A_2 = \{glibc, qt\} \quad (3)$$

Here each of the 155 packages in O_2 depends on *glibc* and *qt*, which are the packages in A_2 . Some of the packages in O_2 also depend on additional packages; however, these dependencies are captured by separate, more specific blocks. Consider $B_3 = (O_3, A_3)$ as an example for a block that is more specific than B_2 . Block B_3 additionally takes the dependency *xorg-x11-libs* into account:

$$O_3 = \{PyQt, arts, doxygen-doxywizard, k3b, kdbg, kdeaddons, \dots, 34 \text{ packages in total}\}$$

$$A_3 = \{glibc, qt, xorg-x11-libs\} \quad (4)$$

Out of the 155 packages that depend on *glibc* and *qt*, only 34 also depend on *xorg-x11-libs*. Note that between B_2 and B_3 the set of dependencies grows ($A_2 \subset A_3$) and the set of packages shrinks ($O_2 \supset O_3$).

FCA records specialization relationships between blocks such as between B_2 and B_3 in a *concept lattice* (Figure 4). We combine this lattice with a statistical analysis to identify dependencies that correlate with vulnerabilities. To assess the vulnerability risk of a block $B = (O, A)$, we measure the percentage of packages in O that are vulnerable, i.e., have a non-zero entry in the vulnerability vector v . This percentage indicates the risk of being vulnerable when depending on the packages in the set A .

$$\text{risk}(B = (O, A)) = \frac{|\{o \mid o \in O, v_o > 0\}|}{|O|}$$

In Figure 4, the risk of B_2 is $120/155 = 77.4\%$ and the risk of B_3 is $27/34 = 79.4\%$. The top most block B_0 in the lattice represents all Red Hat packages because when

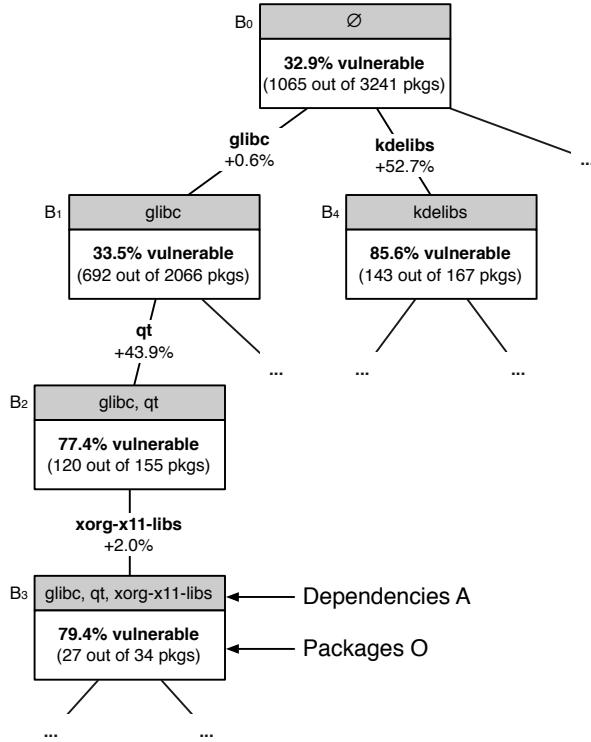


Figure 4: Part of the Red Hat concept lattice.

$A = \emptyset$, every package o satisfies the condition $\forall a \in A : m_{oa} = 1$. Thus when nothing is known about their dependencies the risk of packages is $1065/3241 = 32.9\%$.

Both B_2 and B_3 have a substantially higher risk than B_0 , which means that depending on *glibc*, *qt*, and *xorg-x11-libs* increases the chances of a package being vulnerable. To find out which dependencies matter most, we traverse the concept lattice and test whether the changes in risk are statistically significant. We use χ^2 tests if the entries in the corresponding contingency table are all at least 5, and Fischer Exact Value tests if at least one entry is 4 or less [34, 37].

For example between B_0 and B_1 there is no statistically significant change in risk; in other words, a dependency on *glibc* has little impact on packages being vulnerable. However, risk significantly increases between B_1 and B_2 by 43.9 percentage points, thus depending on *qt* when already depending on *glibc* correlates with the vulnerability of packages. Risk does not change significantly between B_2 and B_3 , which indicates that *xorg-x11-libs* does not increase the risk any further.

Note that in the example, *qt* increases the risk *only* when there is a dependency on *glibc*. We call such a condition the *context C*. The context can be empty; for example in Figure 4, the block B_4 shows that without any context, a dependency on *kdelibs* increases the risk of vulnerabilities by 52.9 percent points.

To find the patterns reported in this paper, we checked for each edge $e = (B_i, B_j)$ in the concept lattice that

- $risk(B_i) \neq risk(B_j)$ at a significance level of $p = 0.01$ (with χ^2 and Fischer Exact Value tests); and
- we have not reported a more general context for the dependency before. For example, if we find that dependency *qt* increases the risk for both contexts $C_1 = \{glibc\}$ and $C_2 = \{glibc, libstdc++\}$, we only report the more general one, which is C_1 in this case.

In total we checked 121,202 edges, for which we found 195 patterns. In this paper, we report only patterns with at least 65 supporting packages (=2% of all Red Hat packages). Table 1 shows the “beast” dependencies that increase the risk of vulnerabilities by at least 20 percent points. In contrast, Table 2 contains the “beauty” dependencies that decrease the risk of vulnerabilities by at least 16.6 percent points.

Several of the beasts in Table 1 are related to security and cryptography, for example, *krb5-libs*, *pam* and *openssl*. One possible reason could be that applications that depend on these packages have to protect sensitive data and thus are more likely to be the target of an attack. Many graphics libraries are beasts as well, for example, *libmng*, *libjpeg*, and *libpng* (related to image file formats) as well as *freetype* and *fontconfig* (related to fonts). Often such libraries are misused by developers and buffer overflows are introduced into an application.

The most notable beauty in Table 2 is *python*, which indicates that Python applications are less likely to be vulnerable. One may ask what about the *perl* package? Here we found two beast rules, which are listed below because they lacked support count to be included in Table 1.

Context	Dep	Context		Context+Dep		Delta
		Count	Risk	Count	Risk	
\emptyset	perl-CGI	3241	0.329	7	0.857	0.529
libxml2	perl	194	0.237	25	0.480	0.243

Applications that depend on *perl-CGI* or use *perl* in addition to *libxml2* are more likely to be exposed to vulnerabilities. However, we advise caution when interpreting these results; Python applications are not guaranteed to be better or safer than Perl applications. Whether an application is vulnerable is not solely determined by dependencies. The experience of developers and the development process are other factors with a strong influence on vulnerabilities.

Another “religious” comparison is between the two rival desktop managers KDE and Gnome. Here, a dependency to *kdelibs* is listed as a beast, while dependencies to *gnome-keyring* and *gnome-libs* are listed as beauties. Again, we advise caution when interpreting these results.

Table 1: The Beasts in Red Hat.

Context	Dependency	Context		Context+Dependency		Delta ≥ 0.200
		Count	Risk	Count ≥ 65	Risk	
\emptyset	openoffice.org-core	3241	0.329	72	1.000	0.671
\emptyset	kdelibs	3241	0.329	167	0.856	0.528
\emptyset	cups-libs	3241	0.329	137	0.774	0.445
\emptyset	libmng	3241	0.329	134	0.769	0.440
glibc	qt	2066	0.335	155	0.774	0.439
glibc	krb5-libs	2066	0.335	108	0.769	0.434
\emptyset	e2fsprogs	3241	0.329	87	0.759	0.430
\emptyset	pam	3241	0.329	116	0.733	0.404
\emptyset	openssl	3241	0.329	313	0.719	0.390
\emptyset	freetype	3241	0.329	251	0.645	0.317
\emptyset	libjpeg	3241	0.329	238	0.639	0.310
\emptyset	gcc-c++	3241	0.329	78	0.628	0.300
\emptyset	libpng	3241	0.329	254	0.626	0.297
\emptyset	libstdc++	3241	0.329	360	0.569	0.241
glibc	fontconfig	2066	0.335	66	0.576	0.241
\emptyset	grep	3241	0.329	66	0.545	0.217
\emptyset	fileutils	3241	0.329	94	0.543	0.214
\emptyset	libgcc	3241	0.329	391	0.535	0.206
<i>(92 rules below threshold)</i>						

Table 2: The Beauties in Red Hat.

Context	Dependency	Context		Context+Dependency		Delta ≤ -0.166
		Count	Risk	Count ≥ 65	Risk	
glibc	xorg-x11-server-Xorg	2066	0.335	66	0.015	-0.320
compat-glibc glibc zlib	audiofile (*)	385	0.613	103	0.359	-0.254
glibc glibc-debug zlib	audiofile (*)	410	0.590	94	0.351	-0.239
\emptyset	gnome-keyring	3241	0.329	69	0.101	-0.227
\emptyset	libglade2	3241	0.329	90	0.111	-0.217
\emptyset	python	3241	0.329	190	0.132	-0.197
XFree86-libs glibc	imlib	493	0.469	103	0.272	-0.197
XFree86-libs glibc glibc-debug	audiofile (*)	397	0.521	104	0.327	-0.194
glibc zlib	libSM	700	0.456	99	0.263	-0.193
glibc zlib	gnome-libs	700	0.456	89	0.281	-0.175
\emptyset	libgnomecanvas	3241	0.329	104	0.154	-0.175
XFree86-libs glibc zlib	audiofile (*)	324	0.531	111	0.360	-0.171
XFree86-libs glibc	esound (*)	493	0.469	114	0.298	-0.170
glibc zlib	libart_lgpl (*)	700	0.456	135	0.289	-0.167
compat-glibc glibc	gnome-libs	1090	0.439	84	0.274	-0.166
<i>(70 rules below threshold)</i>						

Some dependencies are both beasts and beauties, but within different contexts. For example consider the following two rules for the package *esound*:

Context	Dep	Context		Context+Dep		Delta
		Count	Risk	Count	Risk	
glibc2 glibc	esound	312	0.231	45	0.489	0.258
XFree86-libs glibc	esound	493	0.469	114	0.298	-0.170

When applications depend on *glibc2* and *glibc*, an additional dependency to *esound* increases the risk of vulnerabilities. In contrast, when applications depend on *XFree86-libs* instead of *glibc2*, the additional *esound* dependency decreases the risk. Overall, we found only four hybrid dependencies: *audiofile*, *esound*, *glibc*, and *libart_lgpl*. In Table 2, we mark rules involving hy-

brid dependencies with an asterisk (*); there are no such rules in Table 1 because they are below the support count threshold of 65.

Overall, only a few beauties have an empty context, i.e., decrease the risk unconditionally, while most beasts always increase risk. To some extent this is intuitive since any extra dependency adds some potential risk to an application and only a few dependencies have enough positive influence to mitigate this risk. Also, it is important to point out that we reported statistical results. Just adding a dependency to *python* or *gnome-keyring* will not guarantee a safe application. In the end, it is always the developer who introduces a vulnerability, either by using a library incorrectly or by implementing unsafe code.

4 Predicting Vulnerable Packages with SVMs

In the previous section we showed that there is an empirical correlation between certain package dependencies and vulnerabilities. In this section, we use this observation to *predict* which packages have vulnerabilities by using just the names of the dependencies.

We use Support Vector Machines (SVMs) for our prediction models. SVMs are a supervised learning technique that is used for *classification* and *regression* [36]. In the terminology of Section 2, we are given the dependency matrix M and the vulnerability vector v , and the SVM computes a model from them. This is known as *training* the model. Then, one can use this model on a new row vector m_{n+1} to compute a *prediction* \hat{v}_{n+1} . Hatted values such as \hat{v}_k are always the result of a prediction; un-hatted values are known beforehand and are assumed to be exact. Our implementation used the SVM library for the R project [28, 8] with a linear kernel.

We chose SVMs in favor of other machine learning techniques because they have several advantages [15]: first, when used for classification, SVMs cope well with data that is not linearly separable;⁷ second, SVMs are less prone to overfitting.⁸

In order to assess the quality of a classification or regression model, we split the available packages randomly in two parts: a *training set* (two thirds) and a *testing set* (one third). A classification or regression model is then built from the training set and predictions are made for the packages in the testing set. These predictions \hat{v}_k are then compared with the actual observed values v_k and differences are penalized as described in the subsequent sections.

In order to compare the quality of the SVM predictions, we also used decision trees, specifically those resulting from the C4.5 algorithm [27] to train and test the same splits that were used for SVMs. Decision trees are readily interpreted by humans (all classifications can be explained by the path taken in the tree) and therefore have explanatory power that support vector machines lack. It is therefore interesting to compare these two approaches.

4.1 Classifying Vulnerable Packages

For classification, v_k is either 0—no vulnerabilities—or 1—vulnerable. Therefore, the *classification problem* in our case is, “Given new dependency vectors, will their associated vulnerability values be 0 or not?” In other words, given new packages, we want to predict whether they have vulnerabilities or not. A typical use for such a prediction is to assess whether a new package needs

additional testing or review before it is included in a distribution.

For classification, each prediction belongs to one of the following four categories:

- a *true positive* (TP), where $v_k = \hat{v}_k = 1$,
- a *true negative* (TN), where $v_k = \hat{v}_k = 0$,
- a *false positive* (FP), where $v_k = 0$ and $\hat{v}_k = 1$, and
- a *false negative* (FN), where $v_k = 1$ and $\hat{v}_k = 0$.

We want few false positives and false negatives. The two measures used the most to assess the quality of classification models are *precision* and *recall*. They are defined as follows (for both measures, values close to 1 are desirable):

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$
$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

4.2 Ranking Vulnerable Packages

The *regression problem* in our case is, “Given new dependency vectors, what is their rank order in number of vulnerabilities?” In other words, given new packages, we want to know which of them have the most vulnerabilities. A typical use for such a prediction is to decide on the order in which packages are tested or reviewed.

For regression, we report the Spearman rank correlation coefficient ρ , which is a real number between -1 and 1 . If $\rho = 1$, the predicted and actual values have the same ranks (identical rankings): when the predicted values go up, so do the actual values and vice versa. If $\rho = -1$, the predicted and actual values have opposite ranks (opposite ranking): when the predicted values go up, the actual values go down, and vice versa. If $\rho = 0$, there is no correlation between predicted and actual values.

Because the rank correlation coefficient is computed for *all* packages in a testing set, it is an inappropriate measure for how well a model prioritizes resources for quality assurance when only a *subset* of packages are investigated. Let us illustrate this with a simple example. Suppose that we can spend T units of time on testing and reviewing, and that testing or reviewing one package always takes one unit. In the best possible case, our prediction puts the actual top T most vulnerable packages in the top T slots of \hat{v} . However, the *relative order* of these packages does not matter because we will eventually investigate all top T packages. In other words, predicting the actual top T vulnerable packages in any order is acceptable, even though some correlation values ρ will be poor for some of those orderings.

To account for this scenario, we compute an additional measure, which we call *ranking effectiveness*. Let

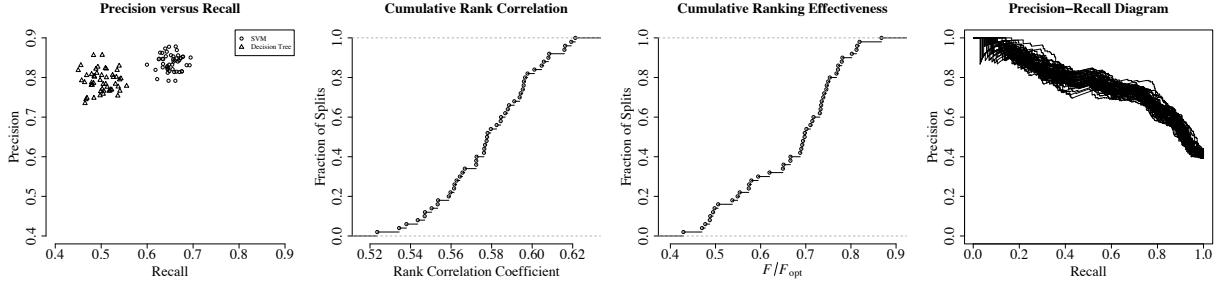


Figure 5: Prediction results for 50 random splits (both classification and ranking).

l be the number of new dependency vectors and let $p = (p_1, \dots, p_l)$ be a permutation of $1, \dots, l$ such that the predictions $\hat{v}_p = (\hat{v}_{p_1}, \dots, \hat{v}_{p_l})$ are sorted in descending order (i.e., $\hat{v}_{p_j} \geq \hat{v}_{p_k}$ for $1 \leq j < k \leq l$). Let q be another permutation that sorts the observed values v_q in descending order. When we now investigate package p_j , by definition we can find and fix v_{p_k} vulnerabilities. Therefore, when we investigate the top T predicted packages, we find

$$F = \sum_{1 \leq j \leq T} v_{p_j}$$

vulnerabilities, but with optimal ordering, we could have found

$$F_{\text{opt}} = \sum_{1 \leq j \leq T} v_{q_j}$$

vulnerabilities. Therefore, we will take the quotient

$$Q = F/F_{\text{opt}} = \sum_{1 \leq j \leq T} v_{p_j} / \sum_{1 \leq j \leq T} v_{q_j} \quad (5)$$

as another quality measure for ranking vulnerable packages.

For ranking, we also report a *precision-recall graph*. This graph visualizes the trade-off between precision and recall by plotting precision against recall when packages are examined in a given order. For effective rankings, the precision will start out near 1.0 and will gradually drop to the fraction of vulnerable packages. Precision-recall graphs indicate how robust prediction models are and can also help choosing a different operating point. For example, in cases where 65% recall is considered to low, a precision-recall diagram allows choosing a higher recall and shows the resulting precision.

4.3 Results

The results of classifying and ranking 50 random splits are shown in Figure 5. The first subfigure is for classification, the others are for ranking.

- *Classification.* For the SVM (shown as circles), the median precision is 0.83 (with a standard deviation of 0.0226), and the median recall is 0.65 (with a standard deviation of 0.0250). This means that our SVM models successfully and consistently catch about two thirds of vulnerable packages and that when a package is predicted as vulnerable, they are correct more than eight times out of ten.

The same figure also contains the respective values for the decision tree (shown as triangles). The median precision is 0.79 (standard deviation 0.0277), and the median recall is 0.50 (standard deviation 0.0264). The median values for both precision and recall are significantly greater for the SVM than for the decision tree models ($p < 0.001$).⁹ The decision tree not only performs worse than the SVM both for precision and recall, the results are also less consistent.

- *Ranking.* The median rank correlation was 0.58 (standard deviation 0.0233), which indicates a consistently moderate to strong correlation; see the second subfigure. The ranking effectiveness values (third subfigure) were computed for $T = 25$ and have a median of 0.70 (standard deviation 0.111), which means that the top 25 predicted packages consistently contain about seventy percent of the maximally possible vulnerabilities.

The last subfigure shows a precision-recall diagram for each of the random splits. These diagrams show the behavior for effective predictors: they start out at or very near to 1.0 and gradually drop to about 0.35, which is the fraction of vulnerable packages ($1133/3241 = 0.35$). The different precision-recall curves also stay close together, indicating consistency across random splits.

5 Discussion

In the previous section we showed that the names of dependencies can actually predict vulnerabilities. In this

section, we refine these results motivated by two observations:

1. If developers know which dependency of a package is most likely to increase the risk of having a vulnerability in the future, they can work on shifting the dependency to another, less risky dependency, or on providing the used services themselves. If the package is new, developers can in many cases even *choose* dependencies with little or no cost.
2. When we predict that a package has unknown vulnerabilities, and this prediction is true in most cases, it may be worthwhile to examine the packages in question by testing or review.

In the subsequent subsections, we first describe a technique to find risky dependencies for a given package (Section 5.1) and next introduce two techniques to identify fragile packages, i.e., non-vulnerable packages that likely will turn vulnerable (Sections 5.2 and 5.3).

5.1 Explaining SVM Predictions

As we have seen, SVMs outperform decision trees for our data set. However, unlike decision trees, SVMs do not explain predictions, which makes it hard for developers to comprehend and have confidence in the outcome of SVM predictions. Even when they know that our models are correct in eight out of ten cases, it remains difficult for them to recognize the two cases where the model errs. In order to better explain the decision of an SVM to developers, we describe how to find dependencies that were most influential for the SVM’s decision. Dependencies that led the SVM to classify a package as vulnerable are candidates for removal, replacement, or increased quality assurance.

An SVM model is a hyperplane H in m dimensions, where $m \geq n$ holds to ensure that the data is linearly separable.¹⁰ When used to classify a dependency vector w , which has n dimensions, the vector is first transformed to a vector w' in m dimensions, according to the kernel used. Then the model looks on which side of the hyperplane vector w' lies and returns the respective classification. The dependency that was *most influential* for the classification of a package is that dependency which moved the package the furthest away from the hyperplane, measured by the *distance* of w' to H . One can use this technique also to rank all dependencies of a package.

Assume first that the linear kernel is used for the SVM. This kernel does not introduce any additional dimensions (thus $m = n$) nor does it perform any transformations ($w = w'$). Since the dependency vector w is *binary* (i.e., w_k is either 0 or 1), one way of computing the most influential dependency is first to drop a perpendicular vector

p from w on H . Then s is the dimension of this perpendicular vector p for which $|p_s|$ is a maximum.

If a kernel other than the linear one is used ($m > n$), we can find the most influential dependency as follows. For every component (dependency) k of the vector w for which $w_j = 1$, we create a new, artificial, “flipped” dependency vector f by setting this component to 0:

$$f_k = \begin{cases} 0 & \text{if } k = j; \\ w_k & \text{otherwise.} \end{cases}$$

Then the most influential dependency is the one for which the flipped and transformed dependency vector f' minimizes the distance to the hyperplane H (or even changes the classification from vulnerable to non-vulnerable). We call this technique *bit-flipping*.

As an example, consider the *sendmail* package with 21 dependencies. The distance between its dependency vector and the separating hyperplane is 3.88. The maximum reduction in distance is 0.73 and occurs with the removal of *cyrus-sasl* from the dependencies of *sendmail*. The package *cyrus-sasl* implements the Simple Authentication and Security Layer (SASL) [21], which is used to add authentication to connection-based protocols such as IMAP. The package is one the most popular SASL implementations; however, the high reduction in distance to the separating hyperplane, suggests that replacing the dependency with another SASL implementation (such as GNU SASL [17]) could decrease the risk of vulnerabilities. In any case, one should track patches and vulnerabilities in the *cyrus-sasl* package to check whether they affect *sendmail*.

5.2 Predicting Fragile Packages with SVMs

In order to predict fragile packages, i.e., regular packages that will turn into vulnerable packages, we again used SVMs. We took RHSAs prior to January 2008 to build a model from which we predicted which non-vulnerable packages have yet undiscovered vulnerabilities. We then used RHSAs from January 2008 onwards and additional information to assess the quality of our model. The higher the percentage of correctly predicted packages, the stronger the model.

The basic idea is to learn an SVM regression model for the entire dataset (until January 2008) and then apply the model again to the same data. Packages without vulnerabilities but with predicted vulnerabilities are then considered to be fragile packages. Essentially, we predict the packages that the SVMs fails to describe in its model (and thus having high residuals) to be fragile.

More formally, using the notation of Section 2, we use an SVM to build a regression model from M . We then

Table 3: Predicted packages.

Package	Reported Vulnerability
✓ #1 <i>mod_php</i>	Integration into <i>php</i> [6]
#2 <i>php-dbg</i>	
#3 <i>php-dbg-server</i>	
#4 <i>perl-DBD-Pg</i>	
#5 <i>kudzu</i>	
#6 <i>irda-utils</i>	
#7 <i>hpoj</i>	
#8 <i>libbdevid-python</i>	
#9 <i>mrtg</i>	
✓ #10 <i>evolution28-evolution-data-server</i>	RHSA-2008:0515-7 ^(a)
#11 <i>lilo</i>	
✓ #12 <i>ckernit</i>	Xatrix Advisory #2006-0029 ^(b)
✓ #13 <i>dovecot</i>	RHSA-2008:0297-6 ^(c)
#14 <i>kde2-compat</i>	
#15 <i>gq</i>	
✓ #16 <i>vorbis-tools</i>	Ubuntu Advisory USN-611-2 ^(d)
#17 <i>k3b</i>	
#18 <i>taskjuggler</i>	
✓ #19 <i>ddd</i>	Inspection (see Section 5.2)
#20 <i>tora</i>	
✓ #21 <i>libpurple</i>	RHSA-2008:0297-6 ^(e)
#22 <i>libwvstreams</i>	
✓ #23 <i>pidgin</i>	RHSA-2008:0584-2 ^(f)
#24 <i>linuxwacom</i>	
✓ #25 <i>policycoreutils-newrole</i>	Changelog entry (Section 5.2)
URLs:	
(a) http://rhn.redhat.com/errata/RHSA-2008-0515.html	
(b) http://www.xatrix.org/advisory.php?s=8162	
(c) http://rhn.redhat.com/errata/RHSA-2008-0297.html	
(d) http://www.ubuntu.com/usn/usn-611-2	
(e) http://rhn.redhat.com/errata/RHSA-2008-0297.html	
(f) http://rhn.redhat.com/errata/RHSA-2008-0584.html	

input the dependency vectors of M into the same SVM model to get n predictions $(\hat{v}_1, \dots, \hat{v}_n)$. Next, we consider only the predictions \hat{v}_j for packages with no known vulnerabilities, that is, for which $v_j = 0$. Finally, we sort the \hat{v}_j in descending order. We hypothesize that packages with high \hat{v}_j are more likely to have vulnerabilities discovered in the future.

For the Red Hat data, we have 3241 packages, of which 2181 had no vulnerabilities reported by January 2008. Until August 2008, 73 packages turned vulnerable (or 3.3%). The result of our prediction is a list of 2181 packages, sorted in decreasing order by expected number of vulnerabilities. We want the newly-found vulnerable packages to appear early in this list. The top 25 predictions are shown in Table 3. Packages found to have vulnerabilities after January 2008 are marked with the symbol ✓. In this case, the last column contains a reference to the respective advisory.

For Table 3, we used sources in addition to the official RHAs.¹¹ We marked package *evolution28-evolution-data-server* as vulnerable because the main package, *evolution28*, was affected by RHSA-2008:0515. In addition, we marked *policycoreutils-newrole* because the

Changelog entry for version 1.30.28-1 reads, “Security fixes to run python in a more locked down manner”. This was apparently a pro-active fix, since there seems to have been no exploit.

In Table 3 the top 25 predictions contain 9 packages with newly-found vulnerabilities (36%). Taking into account the low percentage of packages that turned vulnerable (3.3%), our prediction is significantly better than random guesses (at $p < 0.001$). Note that the 36% is a lower bound for the precision because the non-vulnerable packages might yet have undiscovered vulnerabilities.

Manual inspection of DDD. In order to assess our predictions even in the absence of RHAs or other advisories, we selected the *ddd* package [9]. DDD stands for “Data Display Debugger” and is a graphical front-end for text-based debuggers such as *gdb*. The latest version as of this writing is 3.3.9, released on June 24, 2004. The graphics of DDD are implemented using a combination of plain Xlib (the lowest level of graphics programming using the X Window System), a rather low-level GUI toolkit (Xt), with a GUI library (Motif) on top.

When we performed a cursory review of its source code, we almost immediately found a code-injection vulnerability. This vulnerability occurs in *execpty.C*, where a pipe is opened using *popen()*, but the arguments to the shell are not properly quoted, thus allowing for the insertion of extraneous shell commands for anyone with write access to a configuration file. This will make it possible to run arbitrary code for anyone with local access to the machine if the configuration file is not write protected. Such code injection and arbitrary code execution vulnerabilities are typically classified as “moderate” by Red Hat.

Another security code smell occurs in *xconfig.C* and concerns the use of *fgets()*:

```
char buffer[PATH_MAX];
buffer[0] = '\0';
fgets(buffer, sizeof(buffer), fp);
pclose(fp);

int len = strlen(buffer);
if (len > 0 && buffer[len - 1] == '\n')
    buffer[len - 1] = '\0';
```

The C standard guarantees that *buffer* is null-terminated when any characters are read at all, and unchanged when no characters are read. Therefore, in these two cases, *buffer* will always be properly null-terminated. However, if a read error occurs, the contents of *buffer* are “indeterminate” [16, Section 7.19.7.2]. This means that after a read error, it is no longer guaranteed that *buffer* is null-terminated, the *strlen* call could run away, and the subsequent access to *buffer*[*len* - 1] could cause a buffer overflow. The fix is simple: simply exit whenever

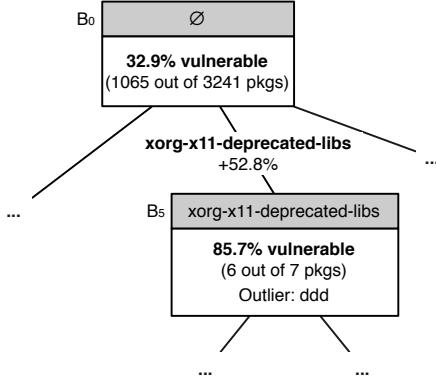


Figure 6: Example of an anomaly.

fgets returns null. The impact of this flaw is not clear; however, there have been arbitrary code execution vulnerabilities resulting from similar errors; see for example CVE 2007-5135 [23].

5.3 Predicting Fragile Packages with Anomalies

Another approach for predicting fragile packages is to search for anomalies in the concept lattice. The basic idea is that for blocks where all but a few packages are vulnerable, the non-vulnerable packages are likely to be fragile. As an example consider $B_5 = (O_5, A_5)$ from Figure 6:

$$\begin{aligned} O_5 &= \{ddd, xpdf, nedit, openmotif, openmotif-devel, \\ &\quad openmotif21, xorg-x11-deprecated-libs-devel\} \\ A_5 &= \{xorg-x11-deprecated-libs\} \end{aligned} \tag{6}$$

All packages in O_5 have been vulnerable, except *ddd*. Thus it is likely that *ddd* soon will have vulnerabilities. We also get a dependency that is responsible for *ddd* being fragile, in this example it is *xorg-x11-deprecated-libs*.

From the 110 rules found in Section 3 for beasts, we selected all rules that had at most three outliers. For the 17 selected rules, we then combined all outliers to come up with a prediction of 27 unique fragile packages (including *ddd*). Out of these, 7 predictions were correct (precision of 25.9%). As in the previous section, the results are significantly better than random guesses ($p < 0.001$) and should be considered a lower bound for precision because of yet undiscovered vulnerabilities.

Manual inspection of DDD. We again inspected DDD, this time with a special focus on the dependency to *xorg-x11-deprecated-libs*. This dependency means that

the package depends on a deprecated Xlib implementation. This is not a vulnerability in itself, but past experience has shown that low-level X Window System programming has been a rich target for exploiters, and deprecated libraries could lack important security fixes more easily than up-to-date ones.

When we look at DDD’s source code, we find this assessment confirmed: much of the graphics code in DDD is on the lowest level, using Xlib directly; other parts use Xt, one of the oldest graphics toolkits for X Windows. This suggests that DDD is old (it was first released in the 1990’s) and has not been actively maintained in the last few years (the most recent release is from 2004). This alone makes it unlikely that it has fixes for all the pitfalls that have surfaced recently. Not surprisingly, the new DDD maintainer wants to switch over to a more modern toolkit such as Qt [9, Entry 2008-05-06], certainly to give DDD a more modern appearance, but also perhaps to offload the burden of maintaining and fixing low-level code to a more active project.

5.4 Threats to Validity

In this section, we discuss threats to validity of our study.

For our analysis, we ignore the possible *evolution of dependencies*. That is, we assume that the dependency matrix M (see Equation 1) does not change with time. We believe that it is reasonable to assume that M will not change much: a dependency of package j on package k exists because package j will want to use services offered by package k . Changing this dependency will mean that the package will have to supply these services itself, stop using them entirely, or use another package to supply them. Any of these alternatives is usually work-intensive, so there is a strong economic incentive against frequently changing dependencies.

One complicating factor when using precision and recall to evaluate our approach is that there may be *undiscovered vulnerabilities* leading to too low values for v_k . For example, it is possible and even likely that for some packages v_k is 0, even though package k does in fact have a vulnerability. In practice, this means that the computed value for the precision will be lower than the true precision value (because the true number of false positives may be lower than what was computed). We can therefore regard our precision values as a lower limit. We cannot make a similar estimation for the recall values, since both false-positive and false-negative values can change. Therefore, the recall values are merely approximations to their true values.

In determining the most influential dependency, we ignore the *joint effect of two or more dependencies*: it could be that two dependencies together are much more influential than a single dependency, and that two together

are a better explanation of the classification of a package than the single dependency that results from the distance minimization technique. This should be the object of further study.

For this paper, we considered only how first-order¹² (or direct) dependencies influence vulnerability. We also did not distinguish between different types and severities of vulnerabilities. In practice, however, many other factors such as developer experience, quality assurance, complexity of source code, and actual usage data likely influence the number of vulnerabilities as well, either separately or in combination. However, there is little scientific evidence for this wisdom and more empirical studies are needed to learn more about vulnerabilities. This paper is a first step in this direction.

6 Possible Interpretations

We described the phenomenon that some dependencies increase vulnerability, and some decrease vulnerability. We also demonstrated that dependencies have predictive ability. Why is this the case?

Our first hypothesis is that dependencies describe the *problem domain* of packages and that some domains are simply more risky than others. For example, we would expect web applications to have more vulnerabilities than C compilers because they have a much larger attack surface. Schröter et al. found similar evidence for the increased error-proneness of some domains [31].

Our second hypothesis is that certain usages may make a package more vulnerable. For example, some packages *use unsafe services*, i.e., services that are inherently unsafe. Similar, there can be also *unsafe use of services*, i.e., some services are difficult to use safely. Both these situations reflect in the dependencies of a package. In an earlier study, Neuhaus found evidence for unsafe usages on the source-file level of Firefox [24].

We will investigate both hypotheses in future work.

7 Related Work

Only few empirical studies exist for software vulnerabilities. Shin and Williams [32] correlated several complexity measures with the number of security problems, for the JavaScript Engine of Mozilla, but found only a weak correlation. This indicates that there are further factors that influence vulnerabilities, like dependencies as we have showed in this paper.

Gegick et al. used code-level metrics such as lines of code, code churn, and number of static tool alerts [12] as well as past non-security faults [11] to predict security faults. In the most recent work, Gegick et al. achieved a precision of 0.52 and a recall of 0.57. In comparison, the

precision and recall values are higher in our experiments (0.83 and 0.65 respectively). However, these numbers are not directly comparable because different data sets were used for the experiments

Based on a pilot study by Schröter et al. [31], Neuhaus et al. [25] investigated the Mozilla project for the correlation of vulnerabilities and *imports*, that is, the *include* directives for the functions called in a C/C++ source file. They found a correlation and were subsequently able to predict with SVMs vulnerabilities that were unknown at the time of the prediction.

Compared to the earlier work by Neuhaus et al. [25], we introduce in this paper an approach to assess the risk of dependencies (concept analysis + statistical testing), compare multiple prediction models (not just SVMs, but also decision trees and anomalies), and show how to explain SVM predictions. Also the *focus of this paper is entirely different*. Instead of a single program, we analyze vulnerabilities for a large software distribution, Red Hat Linux, that consists of several thousand packages. Thus our base of evaluation is much broader: a software distribution covers a wider range of application scenarios, programming languages, and probably every other distinguishing variation, as opposed to a single program. In addition, a software distribution will typically cover a greater range of software quality than a single software project, where the number of contributors is much smaller. The extent of these difference is probably best emphasized by the list of beauties and beasts that we presented in Section 3. This list can serve as a catalog for developers to assess the risk of dependencies and help them make well-informed design decisions.

The idea of finding anomalies using concept analysis (used in Section 5.3) was proposed by Lindig [19]. For the experiments in this paper, we extended Lindig's approach with statistical hypothesis testing. That is, we considered only anomalies for rules which significantly increased the risk of vulnerabilities. In our experiments, this enhancement substantially reduced the number of false positives.

Robles et al. [30] and German [13] studied software distributions to better understand open-source software development. Both studies, however, ignored the relation between package dependencies and vulnerabilities.

Ozment et al. [26] and Li et al. [18] have studied how the number of defects and security issues evolve over time. The two studies report conflicting trends. Additionally, neither of the two approaches allow mapping of vulnerabilities to packages or predictions. Di Penta et al. [7] tracked vulnerabilities across versions in order to investigate how different kinds of vulnerabilities evolve and decay over time.

Alhazmi et al. use the rate at which vulnerabilities are discovered to build models to predict the number of fu-

ture vulnerabilities [2]. In contrast to our approach, their predictions depend on a model of how vulnerabilities are discovered. Tofts et al. build simple dynamic models of security flaws by regarding security as a stochastic process [35], but they do not make specific predictions about vulnerable packages. Yin et al. [38] highlight the need for a framework for estimating the security risks in large software systems, but give neither an implementation nor an evaluation.

8 Conclusion and Consequences

In this paper, we presented a study of vulnerabilities in 3241 software packages of the Red Hat Linux distribution. We provided empirical evidence for a correlation between vulnerabilities and certain dependencies. Furthermore, we showed that prediction models using package dependencies perform well when predicting vulnerabilities. Another observation is that the popular wisdom that vulnerable packages will tend to develop even more vulnerabilities does not hold for the packages within Red Hat: the number of vulnerable packages needing two fixes or fewer (584) is greater than the number of packages needing more than two fixes (549). If the popular wisdom were correct, one would see a majority of packages with a high number of fixes.

Our future work will include the following:

- We will work on refining the distance-minimization technique, looking at how joint effects of dependencies explain SVM predictions.
- We will investigate how the correlation between dependencies and vulnerabilities changes over time. Some beasts will likely become less risky because developers learn from past mistakes. At the same time, new mistakes will likely lead to new beasts.
- We plan to apply our approach to other domains, which require quality assurance; for example Apple’s App Store. Applications undergo a review process before they can be downloaded from the App Store. Using (past) quality and dependency information, Apple could focus on applications that need the most reviewing.
- We want to investigate what other factors predict software vulnerabilities. This paper is just a first step and more empirical studies are needed to better understand security problems.

Often empirical findings are highly project-specific and rarely apply to other projects. This dilemma is illustrated best by a study of Nagappan et al. [22] who compared five large subsystems of Microsoft Windows and found

that for each subsystem, there were metrics that worked reasonably well, but that no single metric worked well for every subsystem to predict failures. Since any empirical study depends on a *large number of context variables* [4], replication has become an important practice to generalize results.

We believe that the work presented in this paper is a first step towards a new generation of empirical studies. Rather than just a few projects, we analyzed vulnerabilities for *several thousand Red Hat packages*. Our findings come therefore with a higher generality compared to traditional single-project studies. While it may be that our work does not generalize to other package collections, we consider this highly unlikely, at least for Linux: other package collections will contain much the same packages, with much the same dependencies. Another characteristic of our study is that *software developers can directly benefit by the results*. By consulting the catalog of beauties and beasts, developers can quickly assess the risk of dependencies to other packages and thus make informed decisions. This lookup is possible with little data (only the dependencies are needed) and without adjusting any prediction models.

To conclude, we are confident that the availability of cross-project repositories (such as the Red Hat Security Advisory database) will lead to more large-scale studies such as the one presented in this paper.

Acknowledgments. Both authors thank Josh Bressers for his valuable help and feedback on this project. We also thank Red Hat for having made their data publicly available and therefore having made this research possible. Thanks to Rebecca Aiken, Ira Cohen, Ahmed Hassan, Andreas Zeller, and the anonymous reviewers for valuable and helpful suggestions on earlier revisions of this paper.

Stephan Neuhaus was supported by funds from the European Commission (contract N° 216917 for the FP7-ICT-2007-1 project MASTER). This work was conducted while Tom Zimmermann was with the Department of Computer Science at the University of Calgary, Canada. Tom Zimmermann was supported by a start-up grant from the University of Calgary.

Notes

¹Software packages are sets of related files, e.g., libraries or applications, distributed in a special file format (RPM) that allows for their automated management, for example through installation and deinstallation.

²For the study in this paper, we consider only packages that are available *from* Red Hat itself because they are the ones supported by Red Hat with security advisories. The total number of RPMs available *for* Red Hat includes third-party RPMs and is thus certainly much larger than 3241.

³Strictly speaking, the security issues addressed in RHSAs need not be vulnerabilities—a vulnerability is considered to be a “*flaw in software that can be exploited*” [33, p. 52]. From looking at a sample of RHSAs, we conclude however that this is almost always the case and thus RHSAs are a good approximation for true vulnerabilities. Josh

Bressers of the Red Hat Security Response Team also confirmed that flaws that do not cross a trust boundary are classified as bugs and not as security advisories [6].

⁴The first RHSA we consider is RHSA-2000:001 and the last is RHSA-2008:0812. When the first advisory was issued in 2006, Red Hat switched to four-digit serial numbers. The serial number at the end is also incremented for bug fix advisories (RHBA) and enhancement advisories (RHEA). Every year the serial number is reset to 0001.

⁵Although version information is also present for each dependency (in the tag `RPMTAG_REQUIREVERSION`), we assumed dependencies to be constant in our experiments. We discuss this decision as a potential threat to validity in Section 5.4.

⁶Formal concept analysis (FCA) is similar to *market basket analysis* or *frequent pattern mining* [1, 20], which made the famous discovery that diapers and beer are often purchased together. In data mining, the set A is called a pattern (for example, diapers and beer) and the set O are the supporting transactions, with $|O|$ being the support count. If $|O|$ exceeds a given threshold, the pattern A is called frequent. FCA additionally provides a lattice with the relations between patterns, which we use to identify dependencies that significantly increase the risk of vulnerabilities.

⁷Two sets of n -dimensional points are said to be *linearly separable* if there exists an $(n - 1)$ -dimensional hyperplane that separates the two sets.

⁸*Overfitting* often happens when a statistical model has too many parameters. The model will try to minimize the error for the training set, but the parameters will offer too many, wildly differing combinations that will make the error small. Choosing one such combination will then generally increase the error for the testing set. The only possible remedy for traditional models is to decrease the number of parameters. SVMs are less prone to overfitting because they choose a specific hyperplane (*maximum margin hyperplane*) among the many that separate the data [36].

⁹The p-value has been corrected for multiple hypothesis testing using the Bonferroni method.

¹⁰Recall that n is the dimensionality of the input space, in our case the number of dependencies.

¹¹In using additional sources, we are not suggesting that Red Hat is negligent in assigning RHSAs. It may well be that the additional advisories found by us are not applicable to Red Hat distributions. Still, security advisories, even when they are not directly applicable to Red Hat packages, indicate that investigating those packages would have been worthwhile.

¹²If package p depends on package q , we call q a *first-order* dependency. If p depends only indirectly on q , we call q a *higher-order* dependency.

References

- [1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'94: Proc. of 20th Int'l. Conf. on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.
- [2] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. *Security Vulnerabilities in Software Systems: A Quantitative Perspective*, volume 3645/2005 of *Lecture Notes in Computer Science*, pages 281–294. Springer Verlag, Berlin, Heidelberg, August 2005.
- [3] Edward C. Bailey. Maximum RPM: Taking the red hat package manager to the limit. <http://www.rpm.org/max-rpm/>, 2000. Last accessed on August 22, 2008.
- [4] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Trans. Software Eng.*, 25(4):456–473, 1999.
- [5] Ladislav Bodnar. Is RPM doomed? <http://distrowatch.com/dwres.php?resource=article-rpm>, 2002. Last accessed: Aug. 2008.
- [6] Josh Bressers. Personal communication, March 2008.
- [7] Massimiliano Di Penta, Luigi Cerulo, and Lerina Aversano. The evolution and decay of statically detected source code vulnerabilities. In *Proc. Int'l. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, 2008.
- [8] Evgenia Dimitriadou, Kurt Hornik, Friedrich Leisch, David Meyer, and Andreas Weingessel. r-cran-e1071. <http://mloss.org/software/view/94/>, 2008.
- [9] Free Software Foundation. DDD data display debugger. <http://www.gnu.org/software/ddd/>, August 2008.
- [10] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin, 1999.
- [11] Michael Gegick, Pete Rotella, and Laurie William. Toward non-security failures as a predictor of security faults and failures. In *Proc. Int'l. Symposium on Engineering Secure Software and Systems (ESSoS)*, 2009. To appear.
- [12] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. Prioritizing software security fortification through code-level metrics. In *QoP '08: Proc. of the 4th ACM workshop on Quality of protection*, pages 31–38. ACM, 2008.
- [13] Daniel M. Germán. Using software distributions to understand the relationship among free and open source software projects. In *Proc. Int'l. Workshop on Mining Software Repositories (MSR)*, page 24, 2007.
- [14] Lawrence A. Gordon, Martin P. Loeb, William Lucyshyn, and Robert Richardson. CSI/FBI computer crime and security survey. Technical report, Computer Security Institute (CSI), 2005.

- [15] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2nd edition, 2005.
- [16] Int'l. Organization for Standardization and Int'l. Electrotechnical Commission. ISO/IEC 9899:TC3 committee draft. Technical report, Int'l. Organization for Standardization, September 2007.
- [17] Simon Josefsson. GNU SASL library—libgsasl. <http://josefsson.org/gsasl>, August 2008.
- [18] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, , and Chengxiang Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proc. Workshop on Architectural and System Support for Improving Software Dependability 2006*, pages 25–33, October 2006.
- [19] Christian Lindig. Mining patterns and violations using concept analysis. <http://www.st.cs.uni-sb.de/~lindig/papers/lindig-2007-mining.pdf>, 2007.
- [20] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *KDD'94: AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192, 1994.
- [21] John G. Myers. RFC 2222: Simple authentication and security layer (sasl). <http://www.ietf.org/rfc/rfc2222.txt>, October 1997.
- [22] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proc. 27th Int'l. Conf. on Software Engineering*, New York, NY, USA, May 2005. ACM Press.
- [23] National Institute of Standards. CVE 2007-5135. <http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-5135>, September 2007.
- [24] Stephan Neuhaus. *Repeating the Past: Experimental and Empirical Methods in Software Security*. PhD thesis, Universität des Saarlandes, Saarbrücken, February 2008.
- [25] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proc. 14th ACM Conf. on Computer and Communications Security (CCS)*, pages 529–540, October 2007.
- [26] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proc. 15th Usenix Security Symposium*, August 2006.
- [27] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman Publishers, San Francisco, CA, USA, 1993.
- [28] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.
- [29] RedHat Network. Errata. <https://rhn.redhat.com/errata/>, August 2008.
- [30] Gregorio Robles, Jesús M. González-Barahona, Martin Michlmayr, and Juan Jose Amor. Mining large software compilations over time: another perspective of software evolution. In *Proc. Workshop on Mining Software Repositories*, pages 3–9, 2006.
- [31] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proc. 5th Int'l. Symposium on Empirical Software Engineering*, pages 18–27, New York, NY, USA, September 2006.
- [32] Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *QoP '08: Proc. 4th ACM workshop on Quality of protection*, pages 31–38. ACM, 2008.
- [33] Adam Shostack and Andrew Stewart. *The New School of Information Security*. Pearson Education, Inc., Boston, MA, USA, 2008.
- [34] Sidney Siegel and N. John Castellan, Jr. *Non-parametric Statistics for the Behavioral Sciences*. McGraw-Hill, 2nd edition, 1988.
- [35] Chris Tofts and Brian Monahan. Towards an analytic model of security flaws. Technical Report 2004-224, HP Trusted Systems Laboratory, Bristol, UK, December 2004.
- [36] Vladimir Naumovich Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, Berlin, 1995.
- [37] Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2nd edition, 2004.
- [38] Jian Yin, Chunqiang Tang, Xiaolan Zhang, and Michael McIntosh. On estimating the security risks of composite software services. In *Proc. PASSWORD Workshop*, June 2006.

Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction

Kristis Makris Rida A. Bazzi
Arizona State University
Tempe, AZ
{makristis,bazzi}@asu.edu

Abstract

We propose a new approach for dynamic software updates. This approach allows updating applications that until now could not be updated at runtime at all or could be updated but with a possibly indefinite delay between the time an update is initiated and the time the update is effected (during this period no service is provided). Unlike existing approaches, we allow arbitrary changes to functions active on the stack and without requiring the programmer to anticipate the future evolution of a program. We argue, using actual examples, that this capability is needed to dynamically update common real applications.

At the heart of our approach is a *stack reconstruction* technique that allows all functions on the call stack to be updated at the same time to guarantee that all active functions have the same version after an update. This is the first general approach that maintains both code and data representation consistency for multi-threaded applications. Our system *UpStare* was used to update the PostgreSQL database management system (more than 200,000 lines of code) and apply 5.5 years-worth of updates to the very secure FTP server vsFTPd (about 12,000 lines of code).

1 Introduction

Downtime experienced by applications due to software updates (feature additions, bug fixes, security patches) can be prohibitive for applications with high-availability requirements. Dynamic Software Update (DSU) can help minimize the downtime by allowing applications to be updated at runtime. Instead of completely stopping the application process and then executing the newer version, DSU would only momentarily pause the application while applying the change *in-memory*. A typical dynamic update would consist of: (1) pausing the execution of the old version in a given state, s ; (2) applying a state

mapping function \mathcal{S} to s to obtain a state $\mathcal{S}(s) = s_{new}$ (loading new code segments can be a part of the mapping); and (3) resume execution of the new version using s_{new} as the initial state. In general, a state mapping needs not happen instantaneously and can be done lazily in stages. The state mapping should be safe in that the resulting state s_{new} should be a valid state of the new application (in a sense that we will make precise in Section 2). In general, a valid state mapping is not always possible, and, when it is possible, it is not necessarily possible for all states of the old application.

The dynamic software update problem consists of two components. First, DSU needs to determine the states, or execution points, of the old application for which it is possible to apply a valid update, and, for those states for which a valid update is possible, to determine the state mapping function to effect the update - this is the *update safety problem*. Second, DSU needs to effect the update through a mechanism that maps an old execution state to a new execution state - this is the *update mechanism problem*. In general, the safety problem is undecidable [6]. This implies that, in general, user help is needed to determine safe update points and to specify the state mapping function. Nevertheless, this does not mean that it is not possible to solve the problem automatically or semi-automatically without or with little user help for many practical cases of interest.

Since user help is unavoidable, it is important to provide the user with an update mechanism and safety checks that make it easier to reason about the update. Current DSU mechanisms are limited in their support of the update of *active functions and data structures* and in their support for *immediate updates*. To support the update of functions that are active on the call stack and for the update of stack-resident data structures, current DSU systems require the user to anticipate the future evolution of a program [3, 14]. Immediate updates are not supported by existing DSU systems. An update is immediate if it satisfies: (1) atomicity: before the update only

old code executes and after the update only new code executes; (2) bounded delay: if a valid mapping is known for a given state and the execution is in that state, then the mapping is applied in a bounded amount of time. Atomicity is desirable because it is sufficient to guarantee *logical consistency* [16, 13]: the execution of the application is *indistinguishable* from an execution in which the old version executes for some time, then the new version executes. While bounded wait is not necessary for logical consistency, we argue that for multithreaded applications, immediate updates are needed to provide logically consistent updates *without service interruption*; i.e. the update does not cause the service to be unavailable for an unbounded amount of time.

To address the limitations of current DSU systems, we propose a new DSU mechanism and a new DSU system for C programs. Our system *UpStare* supports the immediate update of functions that are active on the call stack as well as the update of stack-resident data structures without requiring the user to anticipate the future evolution of the program. Our system is also the first system to allow immediate update for multi-threaded as well as multi-process applications. *UpStare* applies source-to-source transformations to make applications dynamically updateable. At the heart of the mechanism is a novel *stack reconstruction* updating mechanism that allows an application to unroll the call stack when an update occurs (while saving all the stack frames) and then reconstitute the call stack by replacing old versions of functions with their updated versions (while at the same time mapping data structures in the old frames to their updated versions). Stack reconstruction guarantees that after an update is applied only new code executes. Mapping to the new state is automated with an effective heuristic: a patch generator produces data transformers for global variables and for local variables of all stack frames, and a default stack execution continuation mapping resumes execution from the new version. These mappings and transformers can be further fine-tuned by the user. *UpStare*'s immediate data mapping *eliminates the need for data wrappers* that are used by other DSU systems [5, 14, 11, 2] to allow updating datatypes. The elimination of data wrappers greatly reduces execution overhead for data intensive applications.

UpStare supports the update of applications anywhere during their execution including multithreaded applications with blocking system calls. This is achieved by inserting update points in long-lived loops and transforming blocking system calls into non-blocking calls. This guarantees that we can update threads and processes *without interrupting service indefinitely*, since we are not constrained by the need for an active function to exit before we can update it as in other DSU systems.

In summary, our immediate update mechanism guar-

antees the following: (1) Representation consistency; (2) Update immediacy for multi-threaded and multi-process applications; (3) High updateability; (4) No data-access indirection.

UpStare is able to update real-world applications of significant size, such as vsFTPd and PostgreSQL, with minimal manual adjustments from the user and with modest overhead. Still our current implementation has some limitations. First, it is not optimized for performance due to a limitation of existing compilers when partitioning code in hot-cold blocks given branch prediction hints. This limitation can lead to large overhead for systems with a small instruction cache and large functions. Second, it does not yet integrate support to automatically transform pointers, which we developed in previous work [10]. Third, it does not support updates of data in shared memory or in-transit data in internal buffers, but this is not a limitation of the approach; it is a limitation of the current implementation. Finally, since our emphasis in this work is on the updating mechanism, we do not provide automatic safety checks through code analysis as in other DSU systems. Adopting such checks would increase the usefulness of *UpStare*.

The rest of the paper is organized as follows. Section 2 introduces the DSU problem. Section 3 describes our DSU system. Section 4 presents our implementation. Section 5 evaluates the performance of our system and analyzes the sources of overhead. Section 6 discusses related work.

2 The Dynamic Software Update Problem

In this section, we reintroduce the dynamic software update problem (DSU), describe some common safety guarantees that are desirable for DSU systems and argue for the need for immediate updates.

2.1 Dynamic Software Update

Given (Π, s) , where Π is program code and s is an execution state, updating Π to Π_{new} , where Π_{new} is a new version of Π , consists of: (1) pausing the execution Π ; (2) applying a state mapping function S to s to obtain a state $S(s) = s_{new}$; and (3) resuming execution of Π_{new} from state s_{new} .

By updating an executing program, we obtain a hybrid execution that in general needs not satisfy the semantics of either the old or the new versions. In general, the desired semantics for the hybrid executions needs to be determined by the user. We say that a state s for program Π is *valid for update* from Π to Π_{new} if there is a state mapping function S that can be applied in state s such that the resulting hybrid execution satisfies the de-

sired semantics. The dynamic software update problem has two aspects:

- Update safety: Identify a valid state s and a corresponding state mapping function.
- Update mechanism: Implement the state mapping function.

Gupta [6] showed that, even for weak requirements on the semantics of the hybrid execution, it is undecidable to determine if a given state s is valid for update from Π to Π_{new} . The problem is related to the problem of identifying semantic differences [7] between two versions of a program. Identifying semantic differences has been studied extensively and is also undecidable although safe approximations are known [8].

So, in general, assistance from the user is required to both identify valid states and guide the state mapping. Nonetheless, there are many situations in which a default state mapping can produce a new state that will satisfy the desired semantics.

2.2 Safety

Given that it is not possible in general to guarantee the safety of updates without user help, it is helpful to provide some restricted safety guarantees that are satisfied by the updated program. The goal is to make it easier for the user to establish that the default mappings result in valid updates and, if they do not, to supplement the state mapping to make it valid. Some useful guarantees are:

1. Type-safety: No old version of code Π should be executed on a newer version of a datatype representation τ' (oldcode-type-safety) and no new version of code Π' should be executed on an older version of a datatype representation τ (newcode-type-safety).

As an example, consider adding in a C struct that contains five fields a new field as the third field listed and properly constructing a new state s_{new} for a variable of this datatype. If code from the old version accessed the newer version of this datatype in s_{new} it would incorrectly access the memory area used by the new field when intending to access the fourth field, and corrupt data.

2. Transaction-safety: Some sections of code are denoted as transactions and are specified by the user to execute completely in the old version or completely in the new version.

Unlike type safety, transaction safety requires user annotations. One way to ensure transaction safety is by prohibiting updates when execution is in such a user specified section. This can be done at runtime by querying if the current state is in a forbidden region, but this is not straightforward to achieve. If a function f is called inside

a transaction and in other parts of the program, then determining the execution state requires knowledge of the stack contents. Alternatively, transaction safety can be ensured at compile time by conservatively estimating update points that will not violate the transactional requirements.

More generally, a DSU system may be able to provide the user with a more flexible notation to specify that an update is not valid in a given state. For example, stating that an update is not allowed if Thread 1 is executing in (say) <functionA,lines 135-160> while Thread 2 is executing anywhere within <functionB> can be sufficient input to a DSU system to apply the update when these threads do not violate this safety constraint.

3. Representation Consistency: Both state and program representation consistency hold. An update guarantees *state representation consistency* if at no time the executing application expects different representations of state (such as global variables or the stack-frame contents). An update guarantees *program representation consistency* if following the update only Π_{new} is executed over the new state s_{new} ; no part of Π is executed again. Representation consistency (state and program) makes it easier to reason about the effects of executing code on the state because Π_{new} and s_{new} in memory match the source code, but it is not an end-goal in itself. The difference between state representation consistency and type-safety is that one could provide type-safety by allowing new and old definitions of a type to be valid simultaneously. For example, one could apply forward and backward datatype transformers [5], but this makes it harder to reason about updated programs. Additionally, it may not be possible to convert a datatype for new code, then backward for old code, and then forward for new code again, since updated types often contain more information than older types and data could be lost.

4. Logical Representation Consistency: An update system provides logical consistency if the hybrid execution is indistinguishable to an outside observer from executions that are obtained with representationally consistent updates [16, 13].

5. Thread-X-safety: An update is thread-X-safe if X-safety is provided in a multithreaded applications. For example, thread-type-safety means that type-safety is provided for a multithreaded application. In general if a DSU system guarantees that X-safety is satisfied for individual threads independently, then thread-X-safety is not necessarily guaranteed.

Our update mechanism provides the user with the ability to initiate a representationally consistent update in any state of the program. The emphasis is on the mechanism though. Determining the validity of a particular state for update requires other analyses [8, 16, 13].

2.3 Immediate Updates

In this section, we introduce immediate updates and argue that they are needed to guarantee that the update of common multithreaded applications is logically consistent and can be achieved without unbounded service interruption. We first introduce the concept of update with bounded delay.

Bounded delay update: If a valid mapping is known for a valid old state s and the application is in state s , a state mapping can be applied without pausing the application for an unbounded amount of time.

An update is *immediate* if it satisfies representation consistency and bounded delay. To understand the need for immediate updates, consider a multithreaded application in which each server thread handles a client connection and threads read/write in a shared data structure after receiving client requests. In general, there might be a long delay between successive client requests.

Now, consider an update that changes the specification of the data structure and how it is accessed and assume a number of connections are active. To effect the update, there are a number of options:

- Do not allow any new connections and wait until all active connections terminate. When all connection terminate, apply the update. This is not a good option because it can result in the service being unavailable for an unbounded amount of time.
- Allow new connections, but using the old version of the code. This can result in the update being indefinitely delayed because the new version may never get to be executed.
- Allow new connections using the new version of the code while connections created with the old version are active (possibly blocked for client input). This is the more interesting case. Once the shared data structure is accessed by threads running the new version, the data representation would have to reflect the semantics of the new version. This means that on the next access by the old version we either violate logical representation consistency or we force the thread running the old version to be transformed to the new version. Since violating logical consistency is not an option, we are left with the need to immediately update the thread running the old version. Otherwise the connection will not be available for its client for an unbounded amount of time.

So, for all cases, the capability to immediately update individual threads is necessary. If multiple threads of the old version are attempting to access the shared data structures, the updated mechanism should support their

collective immediate update. The update mechanism we propose is the first that can support immediate update of single-threaded as well as multi-threaded applications.

3 Dynamic Update System

We describe our proposed update model and how we apply state mappings under this model.

3.1 Update Model

We propose an update model that is more flexible than the update models of existing works in two respects. First, we consider stack frames as part of updateable program state. Stack frames include local variables, formal parameters, and return addresses. Second, we consider the Program Counter as updateable program state. Unlike existing work, we can ensure updates meet the safety guarantees of Section 2 while employing an updating model that can modify all aspects of the old program state s . This means our approach has a wider reach (more old valid states) in applying an update compared to existing work that needs to accept fewer old valid states if it is to meet these safety guarantees.

A program (Π, s) is a pair of program code Π and program state s . Program code Π is a set containing the executable code of all the functions of the program. Program state $s = (h, T_{sf}, T_{PC})$ is a tuple consisting of a set h containing all global variables on the heap, an array T_{sf} of ordered lists sf of stack frames, one for each thread of the program, and an array T_{PC} of Program Counters for each Thread. Each stack frame $f(l, p, ra)$ in sf contains a set l of local variables on the stack, the formal parameters p and the return address ra . We omit the semantics of program execution from this description.

Software updates are effected by replacing Π with Π_{new} , applying a state transformer S to s , and continuing execution from program Π_{new} in state $S(s) = s_{new}$. Dynamic updates take place at *update points*, which are a subset of possible *PC* locations for the program. Our compiler inserts update points automatically when compiling a program to be update-enabled, as we discuss Section 4. The update mechanism allows the state transformer to modify the entire old program state. For example, for each new stack frame $f'(l', p', ra')$ it can add new local variables to produce l' , change function signatures by extending or reducing the formal parameters to obtain the new formal parameters p' , or adjust the return address ra' of a stack frame to continue from a different execution point on the parent stack frame. It can insert new stack frames in $T_{sf'}$ or remove stack frames. It can also set a new Program Counter $T_{PC'}$ for all threads. For example it can set threads to “escape” from execution of a loop or a function.

3.2 Default State Mapping

Default state mappings are needed to reduce the effort required from the user. In general we would hope that the default mapping is what the user desires, but there are no guarantees for that. The user is always given the capability to override default mappings.

Our approach involves an effective heuristic that relies on verification of its validity by the user. We apply data transformers of global variables on the heap h and local variables l of every stack frame T_{sf} , and re-issue function parameters p . Additionally we map execution continuation of return addresses ra and Program Counters T_{PC} . Transformers and mappings are automatically generated, can be overridden by the user, and, for the cases we have tested, they are effective enough and require minimal user involvement.

Datatype updates. When an update is requested, stack frames T_{sf} and program counters T_{PC} of all running threads are saved and the stack is unrolled up to the thread entry-point function. At this point, the entire old state s at the time the update was initiated is available (having just been saved) to systematically produce the new state s_{new} . For every global variable whose datatype τ has changed, a new global variable of the new datatype τ' is allocated in h' . If the datatype is a **struct** or **union** and it has been extended, a transformer copies the old fields (only new fields must be initialized by the user). If the datatype is reduced, the remaining fields are copied with no user assistance. If the variable is an array, a transformer is applied on all array elements. If the datatype change simply extends an array with more elements (e.g. `parseconf_uint_array` in vsFTPd offers more configuration options), a new array with more room is allocated and the values of all old elements are copied.

Stack frames $f'(l', p', ra')$ are reconstructed with a default automatic mapping by copying the old stack frame $f(l, p, ra)$. Local variables l' are grouped into a **struct** and automatically copied from l . Variable additions are treated as new field additions in a **struct** and can be initialized to a default value by a user-supplied stack transformer. Datatype changes of local variables l are mapped in a way similar to global variables h and formal parameters p' are automatically copied from p or further extended by the user.

Execution continuations. Return addresses ra and Program Counters T_{PC} are automatically preserved, and they correspond to continuation points. Continuation points are all points prior to function calls and all update points. That's how execution control flow can descend to reconstruct a callee, or resume a program after an update, respectively. We take the simple approach of assigning unique numeric ids to continuation points in the order they appear in each function body. By default,

```
struct vsf_transfer_ret
vsf_ftpdataio_transfer_file(
    struct vsf_session* p_sess, int remote_fd,
    int file_fd, int is_recv, int is_ascii)
{
    // Continuation point 1
    if (!is_recv) {
        if (is_ascii) {
            // Continuation point 2
            return do_file_send_ascii(p_sess, remote_fd, file_fd);
        } else {
            // Continuation point 3
            return do_file_send_binary(p_sess, remote_fd, file_fd);
        }
    } else {
        // Continuation point 4
        return do_file_recv(p_sess, remote_fd,
                            file_fd, is_ascii);
    }
}
```

(a) vsFTPd v1.2.2

```
struct vsf_transfer_ret
vsf_ftpdataio_transfer_file(
    struct vsf_session* p_sess, int remote_fd,
    int file_fd, int is_recv, int is_ascii)
{
    filesize_t curr_offset;
    filesize_t num_send;

    // Continuation point 1
    if (!is_recv) {
        if (is_ascii || p_sess->data_use_ssl) {
            // Continuation point 2
            return do_file_send_rwloop(p_sess, file_fd,
                                       is_ascii);
        } else {
            // Continuation point 3
            curr_offset =
                vsf_sysutil_get_file_offset(file_fd);
            // Continuation point 4
            num_send = calc_num_send(file_fd, curr_offset);
            // Continuation point 5
            return do_file_send_sendfile(p_sess, remote_fd,
                                         file_fd, curr_offset, num_send);
        }
    } else {
        // Continuation point 6
        return do_file_recv(p_sess, file_fd, is_ascii);
    }
}
```

(b) vsFTPd v2.0.0

Figure 1: Continuation points in vsFTPd.

we map continuation points with the same enumerator in Π and Π_{new} . If the call graph of the application did not change and the loop structure did not change, this mapping is very effective for actual updates. By adjusting a continuation point a user can define how control flow should continue upon returning to a parent stack frame. We have not found it necessary to insert additional continuation points (e.g. one in every basic block).

Figure 1 shows an example of mapping the continuation of `do_file_send_binary` in an update of vsFTPd from v1.2.2 to v2.0.0. Updating this function requires mapping the ra to its parent stack frame `vsf_ftpdataio_transfer_file`. It requires mapping con-

```

upstare_mapping_t mappings_v200[] = {
    { "vsf_ftpdataio_transfer_file",
      "vsf_ftpdataio_transfer_file",
      2, // 2 continuation points are mapped
      { { 3, 5 },
        { 4, 6 }
      }
    },
    { "do_file_send_binary",
      "do_file_send_sendfile",
      5, // 5 continuation points are mapped
      { { 6, 2 },
        { 7, 3 },
        { 8, 4 },
        { 9, 5 },
        { 10, 6 }
      }
    }
};

```

Figure 2: Relevant continuation mapping for an update of `do_file_send_binary` in vsFTPD v1.2.2 to v2.0.0.

tinuation point 3 from v1.2.2 to continuation point 5 in v2.0.0, including supplying the new parameters `curr_offset` and `num_send` (initialized in the stack transformer) to the new version `do_file_send_sendfile`. Without this mapping an update would incorrectly resume from `ra=3` in v2.0.0, which would load `vsf_sysutil_get_file_offset` on the stack, and the old state T_{sf} of callee stack frames of `do_file_send_binary` would not be restored.

Figure 2 shows the relevant declaration of the variable (source code in C) used to express the continuation mapping to update to vsFTPD 2.0.0. There are two mapping points for `vsf_ftpdataio_transfer_file`: 3 maps to 5, and 4 maps to 6. 1 and 2 use the default mapping: they map to their old values of 1 and 2. Also `do_file_send_binary` is replaced with `do_file_send_sendfile` and execution continues from the replaced function at an offset continuation of -4, which means some code from the beginning of `do_file_send_binary` was removed.

Mapping pointers. Mapping pointers of datatypes known at compile-time is straightforward. However, `void*` pointers are cast at runtime to generic datatypes and are harder to map. Support for tracking pointer types at runtime is needed to invoke the appropriate datatype transforms. We have developed this support in previous work [10] and it has low overhead (1-7%), but we do not yet integrate it with UpStare.

4 Implementation

UpStare consists of a compiler to generate updateable programs, a runtime environment for dynamically applying updates, a patch generator, and a dynamic updating tool, as shown in Figure 3. This architecture is similar to those of existing updating systems. The compiler applies high-level, source-to-source transformations that make

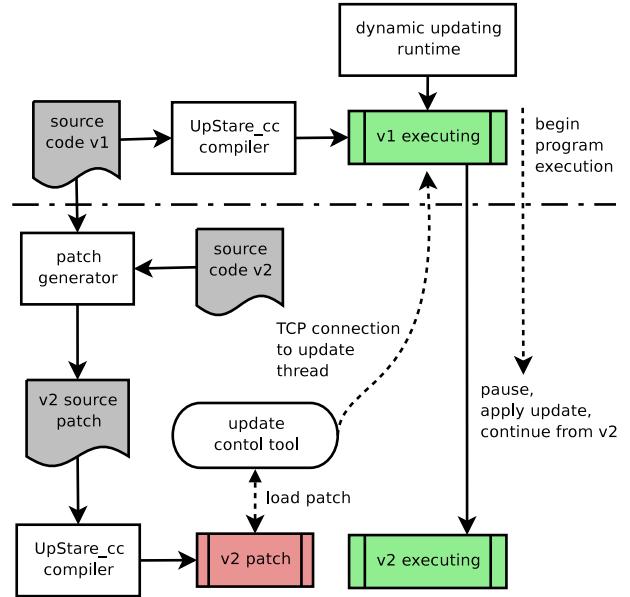


Figure 3: UpStare system architecture.

programs dynamically updateable. It is written in OCaml using the CIL framework[15] v1.3.6 and is architecture and operating system independent. Users replace in their build process (e.g. Makefiles) calls to an existing compiler like `gcc` with calls to the compiler of our system (`hcucc.pl`). No source code modifications by a user are required in existing programs. Programs are transformed as needed to coordinate application of updates with the dynamic updating runtime (written in C; 64KB memory footprint). Updates are initiated by the user with a separate dynamic updating control tool that connects using TCP to a thread waiting for update requests. Updates are loaded in memory using `dlopen` and applied under the guidance of the runtime.

Given the source code of the old and updated programs, a patch generator automatically produces the source code for a dynamic update patch. The patch includes the newer versions of functions, and the old and updated datatype definitions of modified variables, either global or declared on the stack. It also includes automatically generated datatype and stack transformers, and, optionally, user-defined execution continuation mappings that override the default ones to produce the new state.

4.1 Stack Reconstruction

Stack reconstruction consists of two major steps. It saves the existing stack state when unrolling and restores the updated state when reconstructing. To reduce the size of active instrumented code, wrapper functions that efficiently save and restore stack frames are produced away

```

functionA()
{
    char a;
    int param;

    ...
    functionB(param);
}

```

(a) Non-Instrumented

```

typedef struct {
    char a;
    int param;
} stack_functionA_v1_t;

(*functionB_ptr) (int) =
    &functionB_transformed;
functionA_transformed()
{
    stack_functionA_v1_t locals;

    ...
    functionB_6_before:
    functionB_ptr(locals.param);
    if (may_reconstruct && must_reconstruct()) {
        if (must_unroll_up(''functionA'')) {
            save_frame__functionA(&locals, 6);
            return;
        }
        goto functionB_6_before;
    }
}

```

(b) Instrumented

Figure 4: Transformation of function calls for stack reconstruction (`functionB_ptr` just returned).

from the text segment in a separate memory area of cold code executed only during reconstruction.

Figure 4 shows how stack frames are saved. `functionA` is transformed to check upon returning from the callee `functionB` whether the stack should be reconstructed. Note that `may_reconstruct` is a global flag raised only in reconstruction mode to improve performance. If `must_reconstruct` is true (this thread should participate in reconstruction) and execution should be unrolled (`must_unroll_up` is true: the topmost frame, by default, has not been reached yet, but the user can specify that unrolling stops at a different frame), the stack frame and continuation point 6 are saved and `functionA` returns to its caller. Returning to callers continues until the start of the program is reached: the `main` function in single-threaded applications or the start routine passed to a `pthread_create` call for multi-threaded applications. Otherwise unrolling should stop (`must_unroll_up` is false). A `goto` statement resumes execution from `functionB_6_before` and descends in `functionB` for reconstruction.

Figure 5 shows how execution is resumed from `functionA`. If on function entry the stack should be reconstructed downwards, the stack frame is restored. A switch statement maps the continuation point 6 to continuation label `functionB_6_before` using a `goto` state-

```

functionA()
{
    char a;
    int param;

    ...
    functionB(param);
L1:...
}

```

(a) Non-Instrumented

```

functionA_transformed()
{
    stack_functionA_v1_t locals;

    if (may_reconstruct && must_reconstruct()) {
        restore_frame__functionA(&locals);
        switch (next_continuation_point()) {
            ...
            case 3:
                goto try_to_update_3_after;
            ...
            case 6:
                goto functionB_6_before;
            ...
        }
    }
    ...
    functionB_6_before:
    functionB_ptr(locals.param);
    if (may_reconstruct && must_reconstruct()) {
        if (must_unroll_up(''functionA'')) {
            save_frame__functionA(&locals, 6);
            return;
        }
        goto functionB_6_before;
    }
}

```

(b) Instrumented

Figure 5: Transformation of function entrypoints for stack reconstruction (entering `functionA_transformed`).

ment. Execution flow continues by calling `functionB`. When the update is complete (`may_reconstruct` is false: we are no longer in reconstruction mode) and `functionB` finishes, execution continues normally (from L1).

Thread entry-points. If the `main` function or the start routine passed to a `pthread_create` attempt to `return` during reconstruction they will terminate permanently. To allow the update of `main` or thread entry points, calls to such functions are initiated from a wrapper function. To accurately discover thread entry-points (and signal-handlers, discussed next) we use the points-to alias analysis provided by CIL.

Signal handlers. The address of signal handlers, defined with `sigaction` and `signal`, is stored inside the operating system. To avoid resetting signal handlers when they are updated calls to them are initiated from a wrapper function. Additionally, signal handlers return execution to the kernel and are incompatible with stack reconstruction. They are instrumented to raise a flag on entry and reset the flag before exiting. Requests to update are rejected when a program is executing a signal handler.

```

functionA()
{
    char a;
    int param;

    while(condition)
    {
        ...
    }
}

(a) Non-Instrumented

functionA_transformed()
{
    stack_functionA_v1_t locals;

    ...
    while(condition)
    {
        if (must_update) {
            coordinate_update_top(&locals, 3);
            return;
        try_to_update_3_after:
            coordinate_update_bottom();
        }
        ...
    }
}

```

(b) Instrumented

Figure 6: Insertion of an update point at the beginning of a loop.

They are immediately satisfied when the program continues in normal execution mode, and can update signal handlers at that point. Signal handlers are discovered using points-to alias analysis provided by CIL.

Redirecting function calls. Function calls are executed using pointer indirection. For each function `f_v1`, a global variable `f_ptr` is created that points to `&f_v1` and calls to `f_v1` are transformed to calls to `*f_ptr`. For each function pointer `*g_v1`, wrapper functions are created that call it.

Inserting update points. Update points are automatically inserted at the beginning of each function and each loop so they can be encountered often to allow immediate updates. Figure 6 shows an example update point inserted at the beginning of a loop. When the `must_update` flag is raised, the current thread participates in synchronization to block all threads. The current continuation point 3 and the stack frame of `functionA` are saved, and execution returns to the function's caller. When the stack is reconstructed and `functionA` is called again (see Figure 5b), execution flow resumes from `try_to_update_3_after`.

Our current implementation is restricted to a coarse-activation of update points using a single `must_update` flag. However, it is straightforward to support more fine-grain selective activation by dynamically disengaging update points. For example, the user could specify when requesting an update that (say) all update points

except 250-259 should effect the update.

Exporting local variables. The `dlopen` library call will successfully load a dynamic update patch only if the patch references global variables. References to variables that were declared local in the original version (using the `static` keyword) are not accessible after dynamic loading, leading to system exceptions when executing state transformers. Our compiler removes the `static` keyword from all local variables and exports them to global.

4.2 Multi-Threaded Updates

Updating a multi-threaded or multi-process application requires all threads to be blocked. If some threads are not blocked the possibility of thread-safety violations remains open.

We adapted an algorithm that blocks all threads in heterogeneous checkpointing for multi-threaded applications[9] to dynamic updates. The idea is to force all but one thread to block when the application must update. The one thread that is not blocked will be the coordinator of the update. It polls the status of the remaining threads until it can tell for sure that all threads are blocked, as defined below.

When a thread reaches an update point and the application must update, it raises a flag indicating that it is *willing to cooperate* on the update and then attempts to acquire a *coordination lock*. The first thread to acquire the coordination lock is the *coordinator* of the update. The coordinator can tell that some threads are blocked if their cooperation flags are raised. But this does not cover all threads. Some threads might be blocked waiting on an application lock owned by a thread that is already willing to cooperate and that is blocked on the *coordination lock*. To that end, the system needs to keep track of the blocking status of various threads. Calls to `pthread_mutex_lock` and `pthread_mutex_unlock` are replaced with wrapper calls to keep track of the blocking status of threads. When a thread attempts to acquire a lock, it adds the lock to a `WANT` list. When the lock is acquired, the lock is removed from the `WANT` list and placed on a `HAVE` list. When the thread releases the lock, the lock is removed from the `HAVE` list.

The coordinator determines that a thread is *really blocked* if:

1. The thread is willing to update;
2. The thread is blocked waiting on a lock owned by another thread that is *really blocked*.

The coordinator keeps on checking the status of the other threads until it can determine that all other threads are *really blocked*, at which time the coordinator initiates the actual update: the stack of each thread is unrolled

and the threads block; all datatypes are transformed; the stacks are reconstructed and the threads block; and, the threads resume executing the updated version.

The algorithm outlined above has been extended to support blocking threads that use semaphores[9], but our current implementation does not yet integrate that capability with the dynamic update system.

Multi-process updates. We extend multi-threaded updates in multi-process applications. `fork` calls are replaced with wrapper calls that maintain a hierarchy of children. This information is used by the parent process, which acts as a central coordinator of the individual update steps, to apply an atomic update among all children: it waits for all threads of all children to block; all stack frames to be unrolled; transforms datatypes; reconstructs stacks; and, releases all children after all their threads are ready to resume execution. `wait` and `waitpid` are also intercepted to cleanup the children hierarchy.

4.3 Blocking System Calls

To enable the runtime to regain execution when an update is initiated, we transform blocking I/O calls into non-blocking calls and we segment write calls into writes of smaller chunks.

Calls to `sendfile`, which is used in vsFTPd for file transfer, are segmented into 256KB chunks. We do not yet implement segmentation for `send` but it should be straightforward to do so. `read`, `recv`, `accept`, and `select` calls are wrapped to check if the file descriptor is set to blocking mode. If it is, the file descriptor is converted to non-blocking mode, the operation is issued, and execution is voluntarily blocked in a manner that allows unblocking: we issue a `select` that includes in its read set the file descriptor of a pipe created by the runtime. If an update must be applied, hence we need to unblock, we write to the pipe to force `select` to return and encounter an update point. A bottom handler executed after the update point resets the file descriptor to blocking mode. To allow state transformation while a blocking system call is issued without corrupting the data buffer of `read` or `recv`, these calls are issued with a buffer allocated on the heap. When the operations complete, the data are copied back to the original buffer. A possible optimization, which we have not yet implemented, is to transform programs to always allocate I/O data buffers on the heap instead of the stack, to avoid copying data back to the buffer when such operations complete.

A more general approach to handling any blocking system call, not just I/O calls, is to always issue the call in a separate thread. This allows the runtime to remain in control and initiate reconstruction even if the system call has not returned yet. Our original implementation of blocking I/O calls followed this approach but was not as

efficient as the self-pipe `select` solution, due to the cost of `pthread_create` (we did not try worker threads).

5 Evaluation

We demonstrate the working of UpStare on three applications. The data-intensive KissFFT, the vsFTPd server, and the PostgreSQL database. We give a detailed analysis of the sources of overhead, such as runtime overhead, memory footprint, and network overhead.

5.1 KissFFT

We compiled (at `-O3`) the KissFFT¹ v1.2.0 Fast Fourier Transform library (1936 lines of code) using float datatypes to be dynamically updateable and performed 100,000 iterations on 20,000 points. This is an application with heavy data access and for which source code instrumented with Ginseng was made available to us. We did not update this application, but we compiled it to be updateable. We used this application to get a better understanding of the sources of overhead introduced by our instrumentation. We ran experiments that selectively omitted parts of the code that UpStare introduces in an application. We measured the time to run this application: (1) using the original compiler, (2) using CIL, (3) when only wrapper functions to save/restore stack frames are produced, (4) when functions were called directly without pointer indirection, (5) when if-statements without a body are inserted for update points (Figure 6), the switch statement prologue (Figure 5b), or upwards stack unrolling (Figure 4b); here we aim to measure the overhead of branch checks when the `must_update` and `must_reconstruct` flags are not raised, and (6) after adding the body of these if-statements.

Figure 7 shows the impact of the presence of reconstruction-aware code in the program. To compare the results we identify the best compiler to use with a non-instrumented KissFFT and the best compiler to use under instrumentation. Given a non-instrumented KissFFT, `gcc 4.1` (GNU C Compiler) is the best compiler and given an instrumented KissFFT the best compilers are `icc 10.1` (Intel C Compiler) for Ginseng and `gcc 3.4` for UpStare, all on a Pentium M. Under this comparison, the best performing Ginseng reports overhead of 149.8% (87.1% for UpStare) and the best performing UpStare reports overhead of 38.2% (179.3% for Ginseng). The overhead of Ginseng stems from accessing data through a versioned pointer indirection instead of accessing them directly. In comparison, the overhead of UpStare is rooted at the increase of function size that

¹<http://sourceforge.net/projects/kissfft>

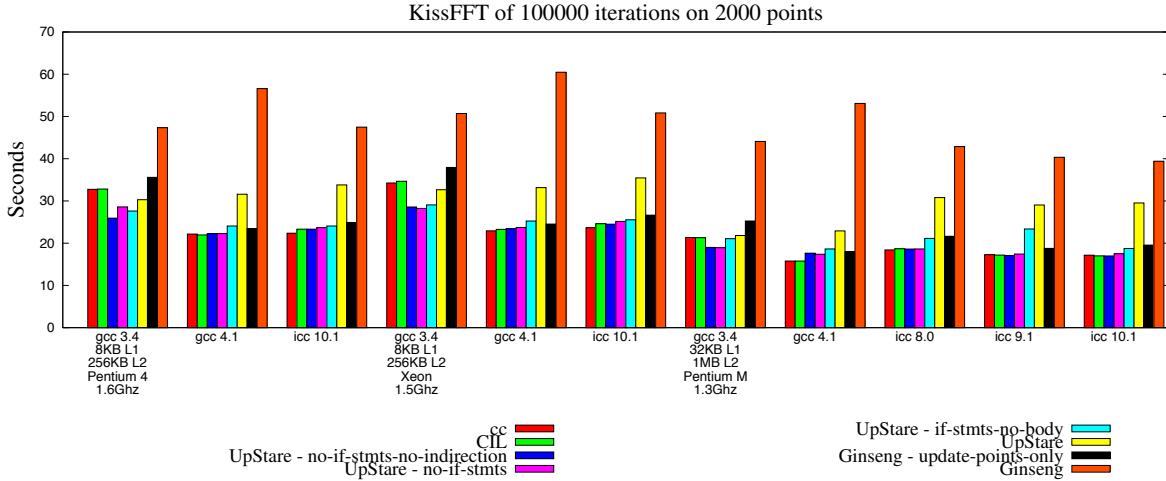


Figure 7: KissFFT: Impact of reconstruction code on running time.

overexerts the ITLB and branch predict unit of the processor.

CIL. CIL transforms source code in simpler terms and should not alter performance. It generally doesn't, but it reported up to 4.2% overhead (Pentium 4: icc 10.1) and up to 1.0% improvement (Pentium M: icc 10.1).

Wrapper save/restore functions. Compared to CIL, producing wrapper functions to save/restore stack frames should not report overhead because these functions are stored outside the text segment. However, on a Pentium M it reported 11.8% overhead with gcc 4.1 and 11.0% improvement with gcc 3.4. Intel compilers report no overhead, suggesting a problem with gcc.

Function indirection. Functions called via pointer indirection should incur constant overhead. They report overhead up to 3.0% on a Pentium M (icc 10.1), 1.2% on a Xeon (gcc 4.1), and 10.3% on a Pentium 4 (gcc 3.4).

If-statements. On a Pentium M, inserting if-statements adds an overhead of 7.2% for icc 10.1, 7.2% for gcc 4.1 and 11.3% for gcc 3.4. This suggests branch prediction can be a significant factor in final performance. Still, update points in Ginseng and UpStare incur comparable overhead.

Increased function size. In comparison to the total overhead of if-statements without a body (Pentium M: 18.0% for gcc 4.1; 9.2% for icc 10.1), an increased function image size adds an overhead of 23.0% and 57.4% respectively, and is responsible for most of the system overhead. We used OProfile to collect performance statistics on the Pentium 4 with gcc 4.1 (overhead 31.3%) and further investigate this issue. We observed a 15% increase in the number of ITLB translations and an 11% increase in the number of instruction fetch requests from the branch predict unit. Other

events like ITLB misses, retired mispredicted branches and page walks showed no significant deviation.

We attempted to use inline assembly to place the body of if-statements outside the text segment. Inline assembly convention prohibits using branch instructions since their presence is not available to high-level optimizations. The compiler would produce intermediate assembly code for the stack unrolling code that would fail to link (inline code supplying linking directives in unreachable basic blocks would not be produced). We also attempted to partition code in hot and cold blocks with `-freorder-blocks-and-partition` using both gcc 4.1 and icc 10.1 but the compilers moved the cold blocks only to the end of the function image without reducing the overhead. Placing the cold blocks to the end of the process image instead may reduce the final overhead.

Memory footprint. We measured the resident set size at the various stages of instrumentation. CIL does not increase the working set. Wrapper code that saves/restores stack frames is responsible for most of the memory increase, up to 236KB (48.7%) using gcc 4.1 on a Pentium M. If-statements marginally increase memory by 4-8KB (0.9-1.7%). The best performing UpStare in respect to running time (Pentium M: gcc 3.4), increased memory by a total of 260KB (53.7%), while Ginseng (Pentium M: icc 10.1) increased memory by 76KB (13.3%). Ginseng increases memory by type wrapping `struct` datatypes, while UpStare adds updateable code inside functions and wrapper functions to save/restore stack frames.

5.2 The Very Secure FTP Daemon

vsFTPd is a fast, secure, widely used FTP application that forks connection handlers that do not communicate

Ver.	Date	LoC ²	Types					Variables					Functions				
			Tot.	Same	Add.	Del.	Upd.	Tot.	Same	Add.	Del.	Upd.	Tot.	Same	Add.	Del.	Upd.
1.1.0	2002-07-31	8,389	628	-	-	-	-	158	-	-	-	-	436	-	-	-	-
1.1.1	2002-10-07	8,468	628	628	0	0	0	161	156	3	0	2	436	420	0	0	16
1.1.2	2002-10-16	8,731	639	626	11	0	2	165	159	4	0	2	447	428	11	0	8
1.1.3	2002-11-09	8,839	646	638	7	0	1	167	164	2	0	1	449	439	2	0	8
1.2.0	2003-05-29	10,011	659	641	16	3	2	201	163	35	1	3	481	378	39	7	64
1.2.1	2003-11-13	10,506	664	655	7	0	2	205	196	7	3	2	486	447	6	1	33
1.2.2	2004-04-26	10,547	664	664	0	0	0	204	202	1	2	1	487	476	1	0	10
2.0.0	2004-07-01	11,527	998	649	342	8	7	218	200	16	2	2	513	421	35	9	57
2.0.1	2004-07-02	11,543	687	674	8	319	5	219	218	1	0	0	513	506	0	0	7
2.0.2	2005-03-03	11,612	688	687	1	0	0	219	219	0	0	0	513	489	1	1	23
2.0.3	2005-03-19	11,743	688	688	0	0	0	226	216	8	1	2	516	481	5	2	30
2.0.4	2006-01-09	11,857	694	687	6	0	1	229	225	3	0	1	519	499	4	1	16
2.0.5	2006-07-03	11,923	694	693	0	0	1	234	228	5	0	1	519	494	0	0	25
2.0.6	2008-02-13	12,202	701	691	7	0	3	239	231	5	0	3	523	497	4	0	22

Table 1: vsFTPd: Source code evolution.

with each other or their parent. We applied 13 updates spanning 5.5 years of application evolution, compiled with gcc 4.1 on a 2.4Ghz Xeon. The updates were prepared automatically using the patch generator. They required a total of 11 user-defined continuation mappings for the two use cases we tested. Additional mappings will probably be needed to update from other update points. They also involved some manual initialization of new variables and `struct` fields.

Table 1 studies the source code evolution of vsFTPd. New datatypes are more often added than modified. Variable additions are common, and there are few datatype changes or variable deletions. Functions are updated very often and are less likely to be deleted. We also note that a large collection of functions and variables are added in major revisions of the program, such as from v1.1.3 to v1.2.0 and from v1.2.2 to v2.0.0. The large number of types added in v2.0.0 is due to including header files from GnuTLS (for secure communication) while in v2.0.1 (released one day later) the OpenSSL header files were removed. We applied updates to vsFTPd under two use cases:

- **Idle client.** A client connected to the server, authenticated correctly, and was waiting idle for user input on the command line. An update was applied.
- **File transfer.** A client connected to the server, authenticated correctly, and requested to retrieve a large file. The file began being transmitted to the client but has not finished transmission. An update was applied.

Our goal was to determine if vsFTPd required updates of functions on the stack under these use cases, which are typical for this type of application. In 7 out of 13 updates the `vsf_session` struct variable allocated in `main` was extended with new fields and needed to be updated. For an idle client, in 7 out of 13 updates functions on the

stack needed to be updated. 5 of those 7 updates were of forward control flow that had not been executed yet and was pending on the stack. For a file transfer, in 9 out of 13 updates functions on the stack needed to be updated and 6 of those 9 updates were of forward control flow. Additionally, we observed a case where an update applied during a large file transfer possibly needed to escape a loop. During the update from v1.1.2 to 1.1.3 the new code in `do_sendfile` should be executed only if a new global flag is on. If the update requires the initial state of this flag to be off, execution should break out of the loop and stop transferring the file.

613 update points where automatically inserted in vsFTPd v2.0.5. Updating during a large file transfer occurred at stack depth 11 (maximum depth is 16, average 8.9) and took 59.7ms: 50.2ms to block all processes; 0.4ms to unroll the stack; 0.95ms to unroll the stack of children processes; 0.45ms to reconstruct; 1ms to reconstruct the stack of children processes. In comparison, Ginseng applies a vsFTPd update in under 5ms [14].

While Ginseng can support the update of `vsf_session` struct, it achieves that with data padding whose limitations we have already discussed.

We setup a client-server configuration connected with a cross-over cable to eliminate network fluctuations. We found this setup necessary to accurately measure performance: in preliminary measurements our system reported performance improvement, which was counter-intuitive. We installed vsFTPd to serve files both from a hard-disk and from an in-memory filesystem to eliminate performance perturbation of hard-disk accesses and identify the worst-case overhead. We measured the latency of establishing a connection and retrieving a 32-byte file 1000 times and the throughput of retrieving a 300MB file. Table 2 reports the median of 11 runs and shows comparable performance for files served either from a hard-disk or from memory. Stack reconstruction slows down an updateable vsFTPd v2.0.5 by ∼0.37–0.50ms (4.9–5.3%), multi-process support by ∼0.65–0.70ms (6.8–7.4%), and support for blocking system calls

²Generated using David A. Wheeler's 'SLOCCount'.

vsFTPd Configuration	Connection Latency(ms) 32-byte file	
	Hard-disk	Memory
v2.0.5 - NonInstrumented	9.61	9.49
v2.0.5 - CIL	9.64 (0.3%)	9.54 (0.5%)
v2.0.5 - Reconstruction	10.08 (4.9%)	9.99 (5.3%)
v2.0.5 - MultiProcess	10.26 (6.8%)	10.19 (7.4%)
v2.0.5 - BlockingCalls	9.97 (3.8%)	9.76 (2.9%)
v2.0.5 - UpStare-FULL	11.15 (16.0%)	11.06 (16.5%)
v2.0.6 - NonInstrumented	9.62	9.52
v2.0.6 - CIL	9.63 (0.1%)	9.54 (0.2%)
v2.0.6 - UpStare-FULL	11.16 (16.0%)	11.09 (16.5%)
v2.0.5 - update to v2.0.6	11.22 (16.6%)	11.12 (16.8%)

Table 2: vsFTPd: Impact of instrumentation on latency.

by $\sim 0.27\text{-}0.36\text{ms}$ (2.9-3.8%). The worst-case overhead is from memory: 1.57ms (16.5%), and 1.63ms (16.8%) when updated to v2.0.6. Ginseng reported overhead of 3% for an updateable and 5% for an updated vsFTPd, but did not report if it eliminated hard-disk accesses or the network from the experiment. In terms of throughput, an updateable v2.0.5 and an update to v2.0.6 reported zero overhead, like Ginseng.

The numbers for latency are presented as a worst-case scenario because, in a practical situation, transferring a file remotely would incur a latency that is considerably larger than the latency of retrieving a 32-byte file. For transferring files, throughput is more relevant and for that measure our system reports zero overhead.

5.3 PostgreSQL Database

PostgreSQL is an advanced DBMS that forks connection handlers that communicate with each other through shared memory. It is a large application of 369K lines of code, with the postmaster process consuming 225K lines of code (source code from `src/backend/`). Using the patch generator, we automatically prepared an update from v7.4.16 to v7.4.17 compiled with gcc 4.1 on a 2.4Ghz Xeon. v7.4.17 updated 64 functions and added one variable. The update was applied dynamically without any user-specified continuation mappings when a client was waiting idle for user input. User-specified mappings will probably be needed to update from other update points (9931 update points where automatically inserted in v7.4.16). The update occurred at stack depth 10 (maximum depth is 35, average 15) and took 60ms: 53.7ms to block all processes; 0.2ms to unroll the stack; 0.45ms to unroll the stack of children processes; 0.3ms to reconstruct the stack; 0.4ms to reconstruct the stack of children processes.

The instrumented v7.4.16 and the update to v7.4.17 passed 85 (out of 93) tests of the PostgreSQL testsuite,

PostgreSQL Configuration	pgbench throughput (t/s) 100,000 transactions	
	Hard-disk	Memory
v7.4.16 - NonInstrumented	175.6	319.7
v7.4.16 - CIL	169.7 (3.4%)	319.0 (0.2%)
v7.4.16 - Reconstruction	133.0 (24.3%)	199.2 (37.7%)
v7.4.16 - MultiProcess	170.5 (2.9%)	312.9 (2.1%)
v7.4.16 - BlockingCalls	161.1 (8.3%)	293.4 (8.2%)
v7.4.16 - UpStare-FULL	130.7 (25.6%)	189.7 (40.7%)
v7.4.17 - NonInstrumented	174.3	317.8
v7.4.17 - CIL	171.3 (1.7%)	316.6 (0.4%)
v7.4.17 - UpStare-FULL	128.0 (26.6%)	189.8 (40.3%)
v7.4.16 - update to v7.4.17	131.8 (24.4%)	188.8 (40.6%)

Table 3: PostgreSQL: Impact of instrumentation on throughput.

both in serial and parallel execution. For the remaining 8 testcases we verified with MPatrol and Valgrind that a non-instrumented PostgreSQL was causing buffer overflows, illegal memory accesses, and uses of uninitialized data. While these access errors seem to produce no problems for an non-instrumented PostgreSQL, they were contributing to failures of other testcases or crashes of a PostgreSQL instrumented with stack reconstruction. Since the memory corruption bugs of PostgreSQL can produce unpredictable results we cannot guarantee our implementation will work in the presence of such bugs.

We measured over a cross-over cable the overhead of an updateable v7.4.16 compared to a non-instrumented v7.4.16 using the PostgreSQL pgbench tool that runs a “TPC-B like” benchmark: five SELECT, UPDATE, and INSERT commands per transaction. We measured the time to run 100,000 transactions after a ramp-up time of 40,000 transactions. Table 3 measures the throughput when the database is loaded both on hard-disk and in memory. Stack reconstruction reports 37.7% overhead in memory but this is a worst-case scenario because a database needs stable storage to be durable (24.3% on hard-disk). Although only one client connection was established overall, multi-process support reported overhead 2.1%-2.9% and blocking system calls 8.3%. An updateable v7.4.16 was 40.7% slower in memory and 25.6% slower on hard-disk. For these cases, the transactions were all executed over the same connection. The numbers show that each transaction consumes 5.7ms and 7.7ms for the non-instrumented and updateable v7.4.16 cases respectively. This translates into a latency overhead of 34.4% for each transaction on average. This latency is for transactions over the same connection.

To measure a worst-case scenario, we measured latency for establishing a connection and running only one transaction over the connection. We measure the latency by running a transaction 1000 times (1000 connections

PostgreSQL Configuration	pgbench latency (ms)	
	Average of 1000 transactions	
	Hard-disk	Memory
v7.4.16 - NonInstrumented	25.62	23.56
v7.4.16 - CIL	25.70 (0.3%)	23.77 (0.9%)
v7.4.16 - Reconstruction	34.98 (36.5%)	33.03 (40.2%)
v7.4.16 - MultiProcess	27.33 (6.7%)	25.44 (8.0%)
v7.4.16 - BlockingCalls	26.94 (5.2%)	25.45 (8.0%)
v7.4.16 - UpStare-FULL	48.09 (87.7%)	45.97 (95.1%)
v7.4.17 - NonInstrumented	25.56	23.53
v7.4.17 - CIL	25.73 (0.7%)	23.64 (0.5%)
v7.4.17 - UpStare-FULL	48.34 (89.1%)	45.85 (94.9%)
v7.4.16 - update to v7.4.17	48.36 (89.2%)	46.21 (96.4%)

Table 4: PostgreSQL: Impact of instrumentation on latency.

were established and torn down). Table 4 reports that the combination of stack reconstruction, multi-process support and blocking system calls support have a severe impact on latency. When isolated, these features report a total overhead of 48.4-56.2%. However, when combined an updateable v7.4.16 is 22.41-22.47ms slower (87.7-95.1%), and 89.2-96.4% slower when updated to v7.4.17. We speculate this is due to the limited size of the processor cache and we intend to run more experiments to better understand the results. Note that the overhead due to reconstruction is comparable to that of KissFFT. We speculate that is due to the nature of the application (data-intensive). We could not obtain a number for Ginseng because it could not compile PostgreSQL but we would expect that the data accesses through pointer indirection in Ginseng would result in high overhead.

6 Related Work

Table 5 compares existing DSU systems with *UpStare*. It first compares kernel updating systems, and then application updating systems.

DynAMOS [11] demonstrates transaction safety through user-supplied adaptation handlers. However it may need to wait indefinitely for a safe update point. Its newcode-type-safety relies on pointer indirection through “shadow data structures”, which incurs overhead, to access the new fields of updated datatypes. But it cannot guarantee oldcode-type-safety if old types change their semantics, like other binary instrumentation systems [17, 2, 1].

K42 [3] is an OS that is particularly crafted to be updateable and its approach cannot be generally applied to existing systems without significant re-engineering. By design it requires all kernel-threads to be short-lived and non-blocking to guarantee *quiescence*: no to-be-updated functions should be active on the stack.

POLUS [4] accomplishes type-safety of global variables by trapping all data accesses for the duration of an update and synchronizing the state of the old and new types. But it cannot update data on the stack, and does not address representation consistency or the thread safety issues of DSU.

Ginseng [14] pads datatypes with enough space to accommodate future growth. Retrieving the appropriate version of padded datatypes during runtime requires indirection for data access. This leads to considerable overhead in data-intensive applications and after many updates there may be no space left to accommodate the update. Ginseng does not provide state and program representation consistency but it offers logical consistency through static analyses [16, 13] which improve safety and updateability. Since its state mapping is restricted, because of its updating mechanism, these conservative analyses may not always find safe update points for that mapping. Still, Ginseng can update multi-threaded applications [12], although continuation may not be immediate. Additionally, Ginseng requires users to anticipate long-lived loops and mark them for “loop extraction” of the loop body into a separate function to update them before the next iteration begins.

Generally, existing systems have difficulty in updating functions [11, 2, 3, 1, 4, 14] and datatypes [1, 3, 4] that are already active on the stack, or function return addresses [17, 11, 2, 3, 1, 4, 14]. They mostly allow functions to be updated the next time they are called [11, 2, 1, 3, 4, 14]. This is due to their restrictive updating mechanism that opens the possibility for executing part old code, part new code, and part old code again, which can be undesirable. Some systems eliminate the possibility of executing mixed code by requiring quiescence before they update [1, 3, 4] but this limits updateability in practice [1, 4]. In Table 5 the overall ability to update from as many old states as possible is coarsely captured in the *updateability* parameter.

UpStare offers high updateability because of its updating mechanism. It can modify all aspects of the old program state (stack-resident functions, datatypes, and return addresses), which allows updating from a wider range of old valid states. Although it provides useful safety guarantees, it requires some involvement from the user in validating semantic safety of updates. *UpStare* has the potential to provide transaction-safety by dynamically disengaging update points, although this is not implemented yet. The transaction safety analysis [13] offered by Ginseng could be used by *UpStare* to reduce user input in validating state transformers.

Acknowledgements. We would like to thank our shepherd George Canea, the anonymous reviewers, and Michael Hicks for their feedback. This work was supported in part by NSF Grant CSR-0849980.

	DynAMOS [11]	K42 [3]	POLUS [4]	Ginseng [14]	UpStare
Domain Preparation	Kernel Binary ✓ Direct Part-indirect High	Kernel Source X Direct Direct Medium	Applications Binary ✓ Direct Trap+Sync Low	Applications Source X Indirect Indirect Low	Applications Source ✓ Direct Direct Medium
No program anticipation by user					
Datatype access					
Updated datatype access					
User involvement for update					
Oldcode type-safety	X	✓	Globals only	Static Analysis	✓
Newcode type-safety	✓	✓	Globals only	Static Analysis	✓
Transaction safety	Adaptive	Quiescence	Quiescence	Static Analysis	Possible
Representation consistency	X	✓	X	X	✓
Logical representation consistency	X	✓	X	✓	✓
Thread safety	X	✓	X	✓	✓
Immediate continuation	X	✓	X	X	✓
Updateability	Medium	High	Low	Medium	High

Table 5: Comparison of existing DSU systems.

References

- [1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online Patches and Updates for Security. In *14th USENIX Security Symposium*, pages 287–302, July 2005.
- [2] Jeff Arnold and M. Frans Kaashoek. KSplice: Automatic Rebootless Kernel Updates. In *EuroSys 2009*, April 2009.
- [3] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Providing Dynamic Update in an Operating System. In *USENIX Symposium on Operating Systems Design and Implementation*, April 2005.
- [4] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.
- [5] Dominic Duggan. Type-based hot swapping of running modules. In *International Conference on Functional Programming*, pages 62–73, 2001.
- [6] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *Software Engineering*, 22(2):120–131, 1996.
- [7] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 234–245, White Plains, NY, June 1990.
- [8] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *In Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411, 1992.
- [9] Feras Karablieh and Rida A. Bazzi. Heterogeneous Checkpointing for Multithreaded Applications. In *21st Symposium on Reliable Distributed Systems (SRDS)*, October 2002.
- [10] Feras Karablieh, Rida A. Bazzi, and Margaret Hicks. Compiler-Assisted Heterogenous Checkpointing. In *20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, October 2001.
- [11] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *EuroSys 2007*, March 2007.
- [12] Iulian Neamtiu. *Practical Dynamic Software Updating*. PhD thesis, University of Maryland, August 2008.
- [13] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 37–50, January 2008.
- [14] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006.
- [15] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compiler Construction*, 2002.
- [16] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2006.
- [17] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Third Symposium on Operating System Design and Implementation*, February 1999.

Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation

Rajesh Krishna Panta, Saurabh Bagchi, Samuel P. Midkiff

School of Electrical and Computer Engineering, Purdue University

{rpanta,sbagchi,smidkiff}@purdue.edu

Abstract

Wireless reprogramming of sensor nodes is an essential requirement for long-lived networks since the software functionality changes over time. The amount of information that needs to be wirelessly transmitted during reprogramming should be minimized since reprogramming time and energy depend chiefly on the amount of radio transmissions. In this paper, we present a multi-hop incremental reprogramming protocol called Zephyr that transfers the *delta* between the old and the new software and lets the sensor nodes rebuild the new software using the received delta and the old software. It reduces the delta size by using application-level modifications to mitigate the effects of function shifts. Then it compares the binary images at the byte-level with a novel method to create small delta, that is then sent over the wireless network to all the nodes. For a wide range of software change cases that we experimented with, we find that Zephyr transfers 1.83 to 1987 times less traffic through the network than Deluge, the standard reprogramming protocol for TinyOS, and 1.14 to 49 times less than an existing incremental reprogramming protocol by Jeong and Culler.

1 Introduction

Large scale sensor networks may be deployed for long periods of time during which the requirements from the network or the environment in which the nodes are deployed may change. This may necessitate modifying the executing application or retasking the existing application with different sets of parameters, which we will collectively refer to as *reprogramming*. Once deployed, it may be very difficult to manually reprogram the sensor nodes because of the scale (possibly hundreds of nodes) and the embedded nature of the deployment since the nodes may be situated in places which are difficult to reach physically. The most relevant form of reprogramming is remote multi-hop reprogramming using the wire-

less medium which reprograms the nodes as they are embedded in their sensing environment. Since the performance of the sensor network is greatly degraded, if not reduced to zero, during reprogramming, it is essential to minimize the time required to reprogram the network. Also, as the sensor nodes have limited battery power, energy consumption during reprogramming should be minimized. Since reprogramming time and energy depend chiefly on the amount of radio transmissions, the reprogramming protocol should minimize the amount of information that needs to be wirelessly transmitted during reprogramming.

In practice, software running on a node evolves, with incremental changes to functionality, or modification of the parameters that control current functionality. Thus the difference between the currently executing code and the new code is often much smaller than the entire code. This makes incremental reprogramming attractive because only the changes to the code need to be transmitted and the entire code can be reassembled at the node from the existing code and the received changes. The goal of incremental reprogramming is to transfer a small *delta* (difference between the old and the new software) so that reprogramming time and energy can be minimized.

The design of incremental reprogramming on sensor nodes poses several challenges. A class of operating systems, including the widely used TinyOS [1], does not support dynamic linking of software components on a node. This rules out a straightforward way of transferring just the components that have changed and linking them in at the node. The second class of operating systems, represented by SOS [6] and Contiki [5], do support dynamic linking. However, their reprogramming support also does not handle changes to the kernel modules. Also, the specifics of the position independent code strategy employed in SOS limits the kinds of changes to a module that can be handled. In Contiki, the requirement to transfer the symbol and relocation tables to the node for runtime linking increases the amount of traffic

that needs to be disseminated through the network.

In this paper, we present a fully functional incremental multi-hop reprogramming protocol called *Zephyr*. It transfers the changes to the code, does not need dynamic linking on the node and does not transfer symbol and relocation tables. *Zephyr* uses an optimized version of the Rsync algorithm [20] to perform *byte-level comparison* between the old and the new code binaries. However, even an optimized difference computation at the low level can generate large deltas because of the change in the positions of some application components. Therefore, before performing byte-level comparison, *Zephyr* performs *application-level modifications*, most important of which is function call indirections, to mitigate the effects of the function shifts caused by software modification.

We implement *Zephyr* on TinyOS and demonstrate it on real multi-hop networks of Mica2 [2] nodes and through simulations. *Zephyr* can also be used with SOS or Contiki to upload incremental changes within a module. We evaluate *Zephyr* for a wide range of software change cases,—from a small parameter change to almost complete application rewrite,—, using applications from the TinyOS distribution and various versions of a real world sensor network application called eStadium [3] deployed at the Ross-Ade football stadium at Purdue University. Our experiments show that Deluge [7], Stream [16], and the incremental protocol by Jeong and Culler [8] need to transfer up to 1987, 1324, and 49 times more number of bytes than *Zephyr*, respectively. This translates to a proportional reduction in reprogramming time and energy for *Zephyr*. Furthermore, *Zephyr* enhances the robustness of the reprogramming process in the presence of failing nodes and lossy or intermittent radio links typical in sensor network deployments due to significantly smaller amount of data that it needs to transfer across the network.

Our contributions in this paper are as follows: 1) We create a small-sized delta for dissemination using optimized byte-level comparisons. 2) We design application-level modifications to increase the structural similarity between different software versions, also leading to small delta. 3) We allow modification to any part of the software (kernel plus user code), without requiring dynamic linking on sensor nodes. 4) We present the design, implementation and demonstration of a fully functional multi-hop reprogramming system. Most previous work has concentrated on some of the stages of the incremental reprogramming system, but not delivered a functional complete system.

2 Related work

The question of reconfigurability of sensor networks has been an important theme in the community. Systems

such as Mate [10] and ASVM [11] provide virtual machines that run on resource-constrained sensor nodes. They enable efficient code updates, since the virtual machine code is more compact than the native code. However, they trade off, to different degrees, less flexibility in the kinds of tasks that can be accomplished through virtual machine programs and less efficient execution than native code. *Zephyr* can be employed to compute incremental changes in the virtual machine byte codes and is thus complementary to this class.

TinyOS is the primary example of an operating system that does not support loadable program modules. Several protocols provide reprogramming with full binaries, such as Deluge [7] and Stream [16]. For incremental reprogramming, Jeong and Culler [8] use Rsync to compute the difference between the old and new program images. However, it can only reprogram a single hop network and does not use any application-level modifications to handle function shifts. We compare the delta size generated by their approach and use it with an existing multi-hop reprogramming protocol to compare their reprogramming time and energy with *Zephyr*. In [19], the authors modify Unix’s diff program to create an edit script to generate the delta. They identify that a small change in code can cause a lot of address changes resulting in a large size of the delta. Koshy and Pandey [9] use slop regions after each function in the application so that the function can grow. However, the slop regions lead to fragmentation and inefficient usage of the Flash and the approach only handles growth of functions up to the slop region boundary. The authors of Flexcup [13] present a mechanism for linking components on the sensor node without support from the underlying OS. This is achieved by sending the compiled image of only the changed component along with the new symbol and relocation tables to the nodes. This has been demonstrated only in an emulator and makes extensive use of Flash. Also, the symbol and relocation tables can grow very large resulting in large updates.

Reconfigurability is simplified in OSes like SOS [6] and Contiki [5]. In these systems, individual modules can be loaded dynamically on the nodes. Some modules can be quite large and *Zephyr* enables the upload of only the changed portions of a module. Specific challenges exist in the matter of reconfiguration in individual systems. SOS uses position independent code and due to architectural limitations on common embedded platforms, the relative jumps can only be within a certain offset (such as 4 KB for the Atmel AVR platform). Contiki disseminates the symbol and relocation tables, which may be quite large (typically these tables make up 45% to 55% of the object file [9]). *Zephyr*, while currently implemented in TinyOS, can also support incremental reprogramming in these OSes by enabling incremental

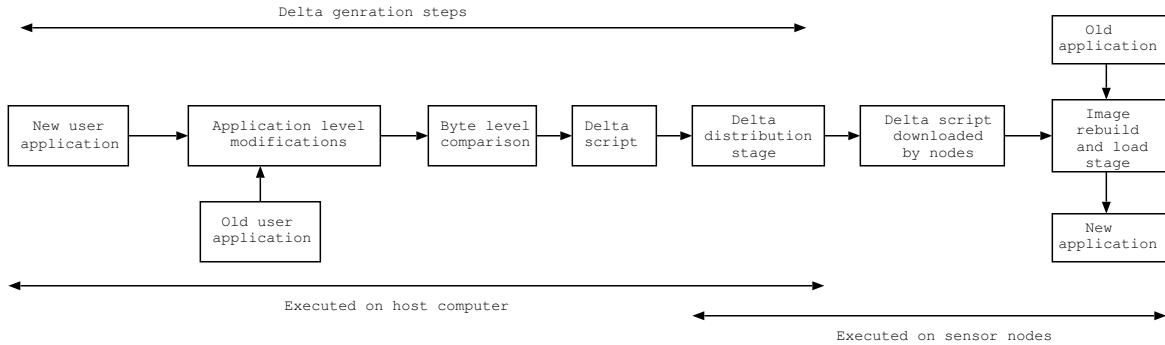


Figure 1: Overview of Zephyr

updates to changed module and updates to kernel modules.

Distinct from this work, in [15], we show that further orthogonal optimizations are possible to reduce the delta size, e.g., by mitigating the effect of shifts of global data variables. One of the drawbacks of Zephyr is that the latency due to function call indirection increases linearly with time. This is especially true for sensor networks because typical sensor applications operate in a loop — sample the sensor, perform some computations, transmit/forward the sensed value to other nodes and repeat the same process. In [15], we solve this while loading the newly rebuilt image from the external flash to the program memory by replacing each jump to the indirection table with a call to the actual function by reading the function address from the indirection table. In this way, we can completely avoid the function call latency introduced by Zephyr.

3 High level overview of Zephyr

Figure 1 is the schematic diagram showing various stages of Zephyr. First Zephyr performs application-level modifications on the old and new versions of the software to mitigate the effect of function shifts so that the similarity between the two versions of the software is increased. Then the two executables are compared at the byte-level using a novel algorithm derived from the Rsync algorithm [4]. This produces the delta script which describes the difference between the old and new versions of the software. These computations are performed on the host computer. The delta script is transmitted wirelessly to all the nodes in the network using the delta distribution stage. In this stage, first the delta script is injected by the host computer to the base node (a node physically attached to the host computer via, say a serial port). The base node then wirelessly sends the delta script to all nodes in the network, in a multi hop manner, if required. The nodes save the delta script in their external flash memory. After the sensor nodes complete downloading the delta script, they rebuild the new image using the

delta and the old image and store it in the external flash. Finally the bootloader loads the newly built image from the external flash to the program memory and the node runs the new software. We describe these stages in the following sections. We first describe byte-level comparison and show why it is not sufficient and thus motivate the need for application-level modifications.

4 Byte-level comparison

We first describe the Rsync algorithm [20] and then our extensions to reduce the size of the delta script that needs to be disseminated.

4.1 Application of Rsync algorithm

Rsync is an algorithm originally developed to update binary data between computers over a low bandwidth network. Rsync divides the files containing the binary data into fixed size blocks. Both sender and receiver compute the pair (Checksum, MD4) over each block. If this algorithm is used as is for incremental reprogramming, then the sensor nodes need to perform expensive MD4 computations for the blocks of the binary image that they have. So, we modify Rsync such that all the expensive operations regarding delta script generation are performed on the host computer and not on the sensor nodes. The modified algorithm runs on the host computer only and works as follows: 1) The algorithm first generates the pair (Checksum, MD4 hash) for each block of the old image and stores them in a hash table. 2) The checksum is calculated for the first block of the new image. 3) The algorithm checks if this checksum matches the checksum for any block in the old image by hash-table lookup. If a matching block is found, Rsync compares if their MD4 hash also match. If MD4 also matches, then that block is considered as a matching block. If no matching block is found for either checksum or MD4, then the algorithm moves to the next byte in the new image and the same process is repeated until a matching block is found. Note that if two blocks do not have the same checksum, then MD4 is not computed for that

block. This ensures that the expensive MD4 computation is done only when the inexpensive checksum matches between the 2 blocks. The probability of collision is not negligible for two blocks having the same checksum, but with MD4 the collision probability is negligible.

After running this algorithm, Zephyr generates a list of COPY and INSERT commands for matching and non matching blocks respectively:

```
COPY <oldOffset> <newOffset> <len>
INSERT <newOffset> <len> <data>
```

COPY command copies *len* number of bytes from *oldOffset* at the old image to *newOffset* at the new image. Note that *len* is equal to the block size used in the Rsync algorithm. INSERT command inserts *len* number of bytes, i.e. *data*, to *newOffset* of the new image. Note that this *len* is not necessarily equal to the block size or its multiple.

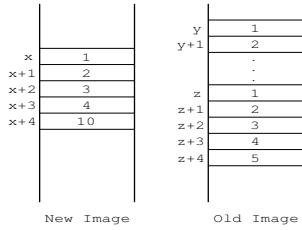


Figure 2: Finding super block

4.2 Rsync optimization

With the Rsync algorithm, if there are *n* contiguous blocks in the new image that match *n* contiguous blocks in the old image, *n* number of COPY commands are generated. We change the algorithm so that it finds the largest contiguous matching block between the two binary images. Note that this does not simply mean merging *n* COPY commands into one COPY command. As shown in Figure 2, let the blocks at the offsets *x* and *x+1* in the new image match those at the offsets *y* and *y+1* respectively in the old image. Let blocks at *x* through *x+3* of the new image match those at *z* through *z+3* respectively of the old image. Note that blocks at *x* and *x+1* match those at *y* and *y+1* and also at *z* and *z+1*. The Rsync algorithm creates two COPY commands as follows: COPY $\langle y \rangle \langle B \rangle \langle x \rangle$ and COPY $\langle y+1 \rangle \langle B \rangle \langle x+1 \rangle$, where *B* is the block size. Then simply combining these 2 commands as COPY $\langle y \rangle \langle 2*B \rangle \langle x \rangle$ does not result in the largest contiguous matching block. The blocks at the offsets *z* through *z+3* form the largest contiguous matching block. We call contiguous matching blocks a *super-block* and the largest super-block the *maximal super-block*. The optimized Rsync algorithm finds the maximal super-block and uses that as the operand in the COPY command. Thus, optimized Rsync produces a single COPY command as COPY $\langle z \rangle \langle 4*B \rangle \langle x \rangle$. Figure

3 shows the pseudo code for optimized Rsync. Its complexity is $O(n^2)$ where *n* is the number of bytes in the image. This is not of a concern because the algorithm is run on the host computer and not on the sensor nodes and only when a new version of the software needs to be disseminated. As we will show in Section 8.2, optimized Rsync running on the desktop computer took less than 4.5 seconds for a wide range of software change cases that we experimented with.

```
/* Terminology
mbl=matching block list
cbl=contiguous block list
*/
1. j=0 and cblStretch=0
2. while j< number of bytes in the new image
3.   mbl=findAllMatchingBlocks(j)
4.   if mbl is empty
5.     j++
6.     if cbl is not empty
7.       Store any one element in cbl as maximum superblock
8.     go to 2
9.   else
10.    j=j+blockSize
11.    if (cblStretch==0)
12.      cbl=mbl
13.      cblStretch++
14.      go to 2
15.    else
16.      Empty tempCbl
17.      for each element in cbl do
18.        if (cbl.element + cblStretch == any entry in mbl )
19.          tempCbl=tempCbl U {cbl.element}
20.      if tempCbl is empty
21.        Store any one element in cbl as maximum superblock
22.        Empty cbl
23.        cblStretch=0
24.      else
25.        cbl=tempCbl
26.        cblStretch++
27.      go to 2
28. end while
findAllMatchingBlocks ( j )
/*Same as Rsync algorithm, but instead of returning the offset
of just one matching block, returns a linked list consisting
of offsets of all matching blocks in the old image for the
block starting at offset j in the new image.*/
```

Figure 3: Pseudo code of optimized Rsync that finds maximal super block

4.3 Drawback of using only byte-level comparison

To see the drawback of using optimized Rsync alone, we consider two cases of software changes.

Case 1: Changing Blink application: Blink is an application in TinyOS distribution that blinks an LED on the sensor node every one second. We change the application from blinking green LED every second to blinking it every 2 seconds. Thus, this is an example of a small parameter change. The delta script produced with optimized Rsync is 23 bytes which is small and congruent with the actual amount of change made in the software.

Case 2: We added just 4 lines of code to Blink. The delta script between the Blink application and the one with these few lines added is 2183 bytes. The actual amount of change made in the software for this case is

slightly more than that in the previous case, but the delta script produced by optimized Rsync in this case is disproportionately larger.

When a single parameter is changed in the application as in Case 1, no part of the already matching binary code is shifted. All the functions start at the same location as in the old image. But with the few lines added to the code as in Case 2, the functions following those lines are shifted. As a result, all the calls to those functions refer to new locations. This produces several changes in the binary file resulting in the large delta script.

The boundaries between blocks can be defined by Rabin fingerprints as done in [18, 14]. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. These fingerprints are efficient to compute on a sliding window in a file. It should be noted that Rabin fingerprint can be a substitute for byte-level comparison only. Because of the content-based boundary between the chunks in Rabin fingerprint approach, the editing operations change only the chunks affected by those edits even if they change the offsets. Only the chunks that have changed need to be sent. But when the function addresses change, all the chunks containing the calls to those functions change and hence need to be sent explicitly. This results in a large delta—comparable to the delta produced by the optimized Rsync algorithm without application-level modifications. Also the *anchors* that define the boundary between the blocks have to be sent explicitly. The chunks in Rabin fingerprints are typically quite large (8 KB compared to less than 20 bytes for our case). As we can see from Figure 6, the size of the difference script will be much larger at 8 KB than at 20 bytes.

5 Application-level modifications

The delta script produced by comparison at the byte-level is not always consistent with the amount of change made in the software. This is a direct consequence of neglecting the application-level structures of the software. So we need to make modifications at the application-level so that the subsequent stage of byte-level comparison produces delta script congruent in size with the amount of software change. One way of tackling this problem is to leave some slop (empty) space after each function as in [9]. With this approach, even though a function expands (or shrinks), the location of the following functions will not change as long as the expansion is accommodated by the slop region assigned to that function. But this approach wastes program memory which is not desirable for memory-constrained sensor nodes. Also, this creates a host of complex management issues like what should be the size of the slop region (possibly different for different functions), what happens if the function expands beyond the assigned slop region, etc. Choosing too large

a slop region means wastage of precious memory and too small a slop region means functions frequently need to be relocated. Another way of mitigating the effect of function shifts is by making the code position independent [6]. Position independent code (PIC) uses relative jumps instead of absolute jumps. However, not all architectures and compilers support this. For example, the AVR platform allows relative jumps within 4KB only and for MSP430(used in TelOS nodes), no compiler is known to fully support PIC.

5.1 Function call indirections

For the byte-level comparison to produce a small delta script, it is necessary to make the adjustments at the application-level to preserve maximum similarity between the two versions of the software. For example, let the application shown in Figure 4-a be changed such that the functions *fun1*, *fun2*, and *funn* are shifted from their original positions *b*, *c*, and *a* to new positions *b'*, *c'*, and *d'* respectively. Note that there can be (and generally will be) more than one call to a function. When these two images are compared at the byte-level, the delta script will be large because all the calls to these functions in the new image will have different target addresses from those in the old image. The approach we take to mitigate the ef-

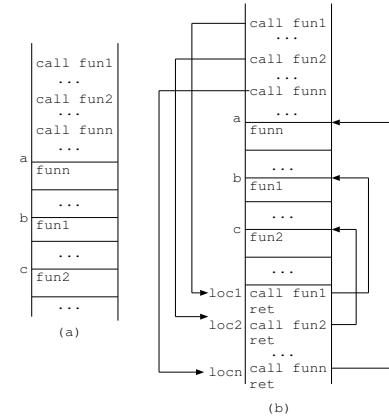


Figure 4: Program image (a) without indirection table and (b) with indirection table.

fects of function shifts is as follows: Let the application be as shown in Figure 4-a. We modify the linking stage of the executable generation process to produce the code as shown in Figure 4-b. Here calls to functions *fun1*, *fun2*, ..., *funn* are replaced by jumps to *fixed locations* *loc1*, *loc2*, ..., *locn* respectively. In common embedded platforms, the call can be to an arbitrarily far off location. The segment of the program memory starting at the fixed location *loc1* acts like an indirection table. In this table, the actual calls to the functions are made. When the call to the actual function returns, the indirection table directs the flow of the control back to the line following the call

to $loc-x$ ($x=1, \dots, n$). The location of the indirection table is kept fixed in the old and the new versions to reduce the size of the delta.

When the application shown in Figure 4-a is changed to the one where the functions $fun1, fun2, \dots, funn$ are shifted, during the process of building the executable for the new image, we add the following features to the linking stage: When a call to a function is encountered, it checks if the indirection table in the old file contains the entry for that function (we also supply the old file (Figure 4-b) as an input to the executable generation process). If yes, then it creates an entry for that function at the indirection table in the new file at the same location as in the old file. Otherwise it makes a decision to assign a slot in the indirection table for that function (call it a *rootless* function) but does not yet create the slot. After assigning slots to the existing functions, it checks if there are any empty slots in the indirection table. These would correspond to functions which were called in the old file but are not in the new file. If there are empty slots, it assigns those slots to the rootless functions. If there are still some rootless functions without a slot, then the indirection table is expanded with new entries to accommodate these rootless functions. Thus, the indirection table entries are naturally garbage collected and the table expands on an as-needed basis. As a result, if the user program has n calls to a particular function, they refer to the same location in the indirection table and only one call, namely the call in the indirection table, differs between the two versions. On the other hand, if no indirection table were used, all the n calls would refer to different locations in old and new applications.

This approach ensures that the segments of the code, except the indirection table, preserve the maximum similarity between the old and new images because the calls to the functions are redirected to the fixed locations even when the functions have moved in the code. The basic idea behind function call indirections is that the location of the indirection table is fixed and hence the target addresses of the jump to the table are identical in the old and new versions of the software. If we do not fix the location of the indirection table, the jump to indirection table will have different target addresses in the two versions of the software. As a result, the delta script will be large. In situations where the functions do not shift (as in Case 1 discussed in Section 4.3) Zephyr will not produce a delta script larger than optimized Rsync without indirection table. This is due to the fact that the indirection tables in the old and the new software match and hence Zephyr finds the large super-block that also contains the indirection table.

The linking changes in Zephyr are transparent to the user. She does not need to change the way she programs. The linking stage automatically makes the above modifi-

cations. Also Zephyr introduces one level of indirection during function calls, but the overhead of function call indirection is negligible because each such indirection takes only few clock cycles (e.g., 8 clock cycles on the AVR platform).

5.2 Pinning the interrupt service routines

It should be noted that due to the change in the software, not only the positions of the user functions but those of the interrupt service routines can also change. Such routines are not explicitly called by the user application. In most of the microcontrollers, there is an interrupt vector table at the beginning of the program memory (generally after the reset vector at 0x0000). Whenever an interrupt occurs, the control goes to the appropriate entry in the vector table that causes a jump to the required interrupt service routine. Zephyr does not change the interrupt vector table to direct the calls to the indirection table as explained above for the normal functions. Instead it modifies the linking stage to always put the interrupt service routines at fixed locations in the program memory so that the targets of the calls in the Interrupt vector table do not change. This further preserves the similarity between the versions of the software.

6 Metacommands for common patterns of changes

After the delta script is created through the above mentioned techniques, Zephyr scans through the script file to identify some common patterns and applies the following optimizations to further reduce the delta size.

6.1 CWI command

We noticed that in many cases, the delta script has the following sequence of commands:

```
COPY <oldOffset=O1> <len=L1> <newOffset=N1>
INSERT <newOffset> <len=l1> <data1>
COPY <oldOffset=O2> <len=L2> <newOffset=N2>
INSERT <newOffset> <len=l1> <data2>
```

and so on. Thus, small INSERT commands would be present in between large COPY commands, e.g., due to different operands op in instruction $ldi r24, op$ commonly found in TinyOS programs while pushing $task$ to the task queue. Here we have COPY commands that copy large chunks of size $L1, L2, L3, \dots$ from the old image followed by INSERT commands with very small values of $len=l1$. Further we notice that $O1+L1+l1=O2, O2+L2+l1=O3$, and so on. In other words, if the blocks corresponding to INSERT commands with small len had matched, we would have obtained a very large superblock. So we replace such sequences with the COPY_WITH_INSERTS (CWI) command.

```
CWI <oldOffset=O1> <newOffset=N1>
<len=L1+l1+...+Ln> <dataSize=l1>
<numInserts=n> <addr1> <data1>
<addr2> <data2> ... <addrn> <datan>
```

Here $dataSize=li$ is the size of $datai$ ($i=1,2,\dots,n$), $numInserts=n$ is the number of $(addr,data)$ pairs, $datai$ are the data that have to be inserted in the new image at the offset $addr_i$. This command tells the sensor node to copy the $len=L1+L2+\dots+Ln$ number of bytes of data from the old image at offset $O1$ to the new image at the offset $N1$, but to insert $datai$ at the offset $addr_i$ ($i=1, 2, \dots, n$).

6.2 REPEAT command

This command is useful for reducing the number of bytes in the delta script that is used to transfer the indirection table. As shown in Figure 4-c and 4-d, the indirection table consists of the pattern $call fun1, ret, call fun2, ret, \dots$ where the same string of bytes (say $S1 = ret; call$) repeats with only addresses for $fun1, fun2$, etc. changing between them. So we use the following command to transfer the indirection table.

```
REPEAT <newOffset> <numRepeats=n>
<addr1> <addr2> ... <addrn>
```

This command puts the string $S1$ at the offset $newOffset$ in the new image followed by $addr1$, then $S1$, then $addr2$, and so on till $addrn$. Note that we could have used the CWI command for this case also. But since the string $S1$ is fixed, we gain more advantage using the REPEAT command. This optimization is not applied if the addresses of the call instructions match in the indirection tables of the old and new images. In that case, COPY command is used to transfer the identical portions of the indirection table.

6.3 No offset specification

We note that if we build the new image on the sensor nodes in a *monotonic* order, then we do not need to specify the offset in the new file in any of the above commands. Monotonic means we always write at location x of the new image before writing at location y , for all $x < y$. Instead of the new offset, a counter is maintained and incremented as the new image is built and the next write always happens at this counter. So, we can drop the $newOffset$ field from all the commands.

We find that for Case 2, where some functions were shifted due to addition of few lines in the software, the delta script produced with the application-level modifications is 280 bytes compared to 2183 bytes when optimized Rsync was used without application-level modifications. The size of the delta script without the meta-commands is 528 bytes. This illustrates the importance of application-level modifications in reducing the size of the delta script and making it consistent with the amount of actual change made in the software.

7 Delta distribution stage

One of the factors that we considered for the delta distribution stage was to have as small a delta script as possible even in the worst case when there is a huge change

in the software. In such a case there is very little similarity between the old and the new code images and the delta script basically consists of a large INSERT command to insert almost the entire binary image. To have small delta script even in such extreme cases, it is necessary that the binary image itself be small. Since the binary image transmitted by Stream [16] is almost half compared to that of Deluge [7], Zephyr uses the approach from Stream with some modifications for wirelessly distributing the delta script. The core data dissemination method of Stream is the same as in Deluge. Deluge uses a monotonically increasing version number, segments the binary code image into pages, and pipelines the different pages across the network. The code distribution occurs through a three-way handshake of advertisement, request, and code broadcast between neighboring nodes. Unlike Deluge, Stream does not transfer the entire reprogramming component every time code update is done. The reason behind this requirement in Deluge is that the reprogramming component needs to be running on the sensor nodes all the time so that the nodes can be receptive to future code updates and these nodes are not capable of multitasking (running more than one application at a time). Stream solves this problem by storing the reprogramming component in the external flash and running it on demand—whenever reprogramming is to be done.

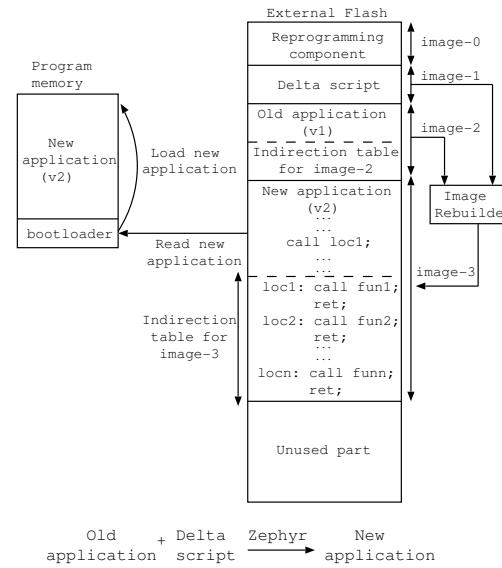


Figure 5: Image rebuild and load stage. The right side shows the structure of external flash in Zephyr.

Distinct from Stream, Zephyr divides the external flash as shown in the right side of Figure 5. The reprogramming component and delta script are stored as image 0 and image 1 respectively. Image 2 and image 3 are the user applications—one old version and the other

current version which is created from the old image and the delta script as discussed in Section 7.1. The protocol works as follows:

- 1) Let image 2 be the current version (v_1) of the user application. Initially all nodes in the network are running image 2. At the host computer, delta script is generated between the old image (v_1) and the new image (v_2).
- 2) The user gives the command to the base node to reboot all nodes in the network from image 0 (i.e. the reprogramming component).
- 3) The base node broadcasts the reboot command and itself reboots from the reprogramming component.
- 4) The nodes receiving the reboot command from the base node rebroadcast the reboot command and themselves reboot from the reprogramming component. This is controlled flooding because each node broadcasts the reboot command only once. Finally all nodes in the network are executing the reprogramming component.
- 5) The user then injects the delta script to the base node. It is wirelessly transmitted to all nodes in the network using the usual 3-way handshake of advertisement, request, and code broadcast as in Deluge. Note that unlike Stream and Deluge which transfer the application image itself, Zephyr transfers the delta script only.
- 6) All nodes store the received delta script as image 1.

7.1 Image rebuild and load stage

After the nodes download the delta script, they rebuild the new image using the script (stored as image 1 in the external flash) and the old image (stored as image 2 in the external flash). The image rebuild stage consists of a *delta interpreter* which interprets the COPY, INSERT, CWI, and REPEAT commands of the delta script and creates the new image which is stored as image 3 in the external flash.

The methods of rebooting from the new image are slightly different in Stream and Zephyr. In Stream, a node automatically reboots from the new code once it has completed the code update and it has satisfied all other nodes that depend on this node to download the new code. This means that different nodes in the network start running the new version of the code at different times. However, for Zephyr, we modified Stream so that all the nodes reboot from the new code after the user manually sends the reboot command from the base station (as in Deluge). We made this change because in many software change cases, the size of the delta script is so small that a node (say n_1) nearer to the base station quickly completes downloading the code before a node (say n_2) further away from the base station even starts requesting packets from n_1 . As a result, n_1 reboots from the new code so fast that n_2 cannot even start the download process. Note that this however does not pose a correctness issue. After n_1 reboots from the new code,

it will switch again to the reprogramming state when it receives advertisement from n_2 . However, this incurs the performance penalty of rebooting from a new image. Our design choice has a good consequence—all nodes come up with the new version of the software at the same time. This avoids the situation where different nodes in the network run different versions of the software. When a node receives the reboot command, its bootloader loads the new software from image 3 of the external flash to the program memory (Figure 5). In the next round of reprogramming, image 3 will become the old image and the newly rebuilt image will be stored as image 2. As we will show in Section 8.3, the time to rebuild the image is negligible compared to the total reprogramming time.

7.2 Dynamic page size

Stream divides the binary image into fixed-sized pages. The remaining space in the last page is padded with all 0s. Each page consists of 1104 bytes (48 packets per page with 23 bytes payload in each packet). With Zephyr, it is likely that in many cases, the size of the delta script will be much smaller than 1104 bytes. For example, we have delta script of sizes of 17 bytes and 280 bytes for Case 1 and Case 2 respectively. Also, as we will show in Section 8.2, during the natural evolution of the software, it is more likely that the nature of the changes will be small or moderate and as a result, delta scripts will be much smaller than the standard page size. After all, the basic idea behind any incremental reprogramming protocol is based on the assumption that in practice, the software changes are generally small so that the similarities between the two versions of the software can be exploited to send only small delta. When the size of the delta script is much smaller than the page size, it is wasteful to transfer the whole page. So, we change the basic Stream protocol to use dynamic page sizes.

When the delta script is being injected to the base node, the host computer informs it of the delta script size. If it is less than the standard page size, the base node includes this information in the advertisement packets that it broadcasts. When other nodes receive the advertisement, they also include this information in the advertisement packets that they send. As a result, all nodes in the network know the size of the delta script and they make the page size equal to the actual delta script size. So unlike Deluge or Stream which transmit all 48 data packets per page, Zephyr transmits only required number of data packets if the delta script size is less than 1104 bytes. Note that the granularity of this scheme is the packet size, i.e., the last packet of the last page may be padded with zeros. But this results in small enough wastage that we did not feel justified in introducing the additional complexity of dynamic packet size. Our scheme can be further modified to advertise the actual number of packets

of the last page. This would minimize the wastage, for example in the case where the delta script has 1105 bytes, it would transfer 2 pages, the first page with 48 packets and the second with 1 packet.

8 Experiments and results

In order to evaluate the performance of Zephyr, we consider a number of software change scenarios. The software change cases for standard TinyOS applications that we consider are as follows:

Case 1: Blink application blinking a green LED every second to blinking every 2 seconds.

Case 2: Few lines added to the Blink application.

Case 3: Blink application to CntToLedsAndRfm: CntToLedsAndRfm is an application that displays the lowest 3 bits of the counting sequence on the LEDs as well as sends them over radio.

Case 4: CntToLeds to CntToLedsAndRfm: CntToLeds is the same as CntToLedsAndRfm except that the counting sequence is not transmitted over radio.

Case 5: Blink to CntToLeds.

Case 6: Blink to Surge: Surge is a multi hop routing protocol. This case corresponds to a complete change in the application.

Case 7: CntToRfm to CntToLedsAndRfm: CntToRfm is the same as CntToLedsAndRfm except that the counting sequence is not displayed on the LEDs.

In order to evaluate the performance of Zephyr with respect to natural evolution of the real world software, we considered a real world sensor network application called eStadium [3] deployed in Ross Ade football stadium at Purdue University. eStadium applications provide safety and security functionality, infotainment features such as coordinated cheering contests among different parts of the stadium using the microphone data, information to fans about lines in front of concession stands, etc. We considered a subset of the changes that the software had actually gone through, during various stages of refinement of the application.

Case A: An application that samples battery voltage and temperature from MTS310 [2] sensor board to one where few functions are added to sample the photo sensor also.

Case B: During the deployment phase, we decided to use opaque boxes for the sensor nodes. So, a few functions were deleted to remove the light sampling features.

Case C: In addition to temperature and battery voltage, we added the features for sampling all the sensors on the MTS310 board except light (e.g., microphone, accelerometer, magnetometer). This was a huge change in the software with the addition of many functions. For accelerometer and microphone, we collected mean and mean square values of the samples taken during a user specified window size.

Case D: This is the same as Case C but with addition of few lines of code to get microphone peak value over the user-specified window size.

Case E: We decided to remove the feature of sensing and wirelessly transmitting to the base node, the microphone mean value since we were interested in the energy of the sound which is given by the mean square value. A few lines of code were deleted for this change.

Case F: This is same as Case E except we added the feature of allowing the user to put the nodes to sleep for a user-specified duration. This was also a huge change in the software.

Case G: We changed the microphone gain parameter. This is a simple parameter change.

We can group the above changes into 4 classes:

Class 1 (Small change SC): This includes Case 1 and Case G where only a parameter of the application was changed.

Class 2 (Moderate change MC): This includes Case 2, Case D, and Case E. They consist of addition or deletion of few lines of the code.

Class 3 (Large change LC): This includes Case 5, Case 7, Case A, and Case B where few functions are added or deleted or changed.

Class 4 (Very large change VLC) : This includes Case 3, Case 4, Case 6, Case C, and Case F.

Many of the above cases involve changes in the OS kernel as well. In TinyOS, strictly speaking, there is no separation between the OS kernel and the application. The two are compiled as one big monolithic image that is run on the sensor nodes. So, if the application is modified such that new OS components are added or existing components are removed, then the delta generated would include OS updates as well. For example, in Case C, we change the application that samples additemperature and battery voltage to the one that samples microphone, magnetometer and accelerometer sensors in addition to temperature and battery. This causes new OS components to be added—the device drivers for the added sensors.

8.1 Block size for byte-level comparison

We modified Jarsync [4], a java implementation of the Rsync algorithm, to achieve the optimizations mentioned in Section 4.2. From here onward, by semi-optimized Rsync, we mean the scheme that combines two or more contiguous matching blocks into one super-block. It does not necessarily produce the maximal super-block. By optimized Rsync we mean our scheme that produces the maximal super-block but without the application-level modifications.

As shown in Figure 6, the size of the delta script produced by Rsync as well as optimized Rsync depends on the block size used in the algorithm. Recollect that the comparison is done at the granularity of a block. As ex-

Table 1: Comparison of delta script size of various approaches. Deluge, Stream and Rsync represent prior work.

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case A	Case B	Case C	Case D	Case E	Case F	Case G
Deluge:Zephyr	1400.82	85.05	4.52	4.29	8.47	1.83	29.76	7.60	7.76	2.63	203.57	243.25	2.75	1987.2
Stream:Zephyr	779.29	47.31	2.80	2.65	4.84	1.28	18.42	5.06	5.17	1.82	140.93	168.40	1.83	1324.8
Rsync:Zephyr	35.88	20.81	2.06	1.96	3.03	1.14	8.34	3.35	3.38	1.50	36.03	42.03	1.50	49.6
SemiOptimizedRsync:Zephyr	6.47	11.75	1.80	1.72	2.22	1.11	5.61	2.66	2.71	1.39	14.368	17.66	1.36	6.06
OptimizedRsync:Zephyr	1.35	7.79	1.64	1.57	2.08	1.07	3.87	2.37	2.37	1.35	7.84	9.016	1.33	1.4
ZephyrWithoutMetacommands:Zephyr	1.35	1.99	1.38	1.30	1.39	1.05	1.52	1.6	1.61	1.16	2.33	2.43	1.18	1.4

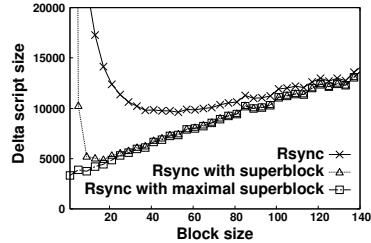
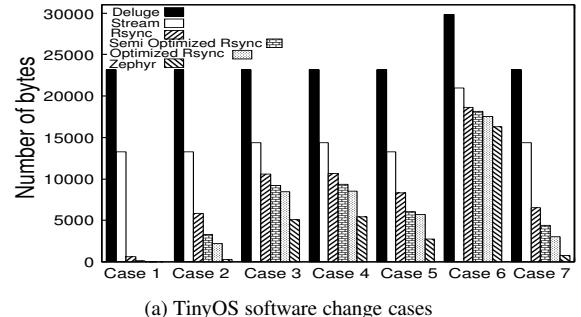


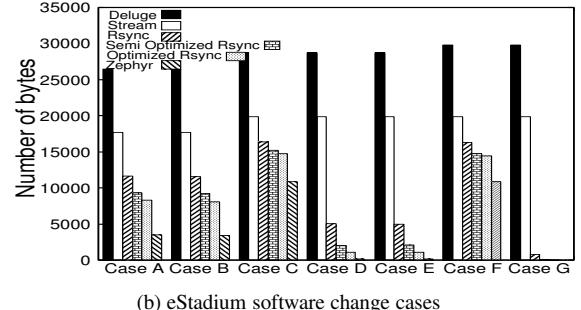
Figure 6: Delta script size versus block size

pected, Figure 6 shows that the size of the delta script is largest for Rsync and smallest for optimized Rsync. It also shows that as block size increases, the size of the delta script produced by Rsync and semi-optimized Rsync decreases till a certain point after which it has an increasing trend. The size of the delta script depends on two factors: 1) number of commands in the delta script and 2) size of data in the INSERT command. For Rsync and semi-optimized Rsync, for block size below the minima point, the number of commands is high because these schemes find lots of matching blocks but not (necessarily) the maximal super-block. As block size increases in this region, the number of matching blocks and hence the number of commands drops sharply causing the delta script size to decrease. However, as the block size increases beyond the minima point, the decrease in the number of commands in the delta script is dominated by the increase in the size of new data to be inserted. As a result, the delta script size increases. For optimized Rsync, there is a monotonic increasing trend for the delta script size as block size increases. There are however some small oscillations in the curve, as a result of which the optimal block size is not always one byte. The small oscillations are due to the fact that increasing the block size decreases the size of maximal super-blocks and increases the size of data in INSERT commands. But sometimes the small increase in size of data can contribute to reducing the size of the delta script by reducing the number of COPY commands. Nonetheless, there is an overall increasing trend for optimized Rsync. This has the important consequence that a system administrator using Zephyr does not have to figure out the block size to use in uploading code for each application change. She can use the smallest or close to smallest block size

and let Zephyr be responsible for compacting the size of the delta script. In all further experiments, we use the block size that gives the smallest delta script for each scheme—Rsync, semi-optimized Rsync, and optimized Rsync.



(a) TinyOS software change cases



(b) eStadium software change cases

Figure 7: Size of data transmitted for reprogramming

8.2 Size of delta script

The goal of an incremental reprogramming system is to reduce the size of the delta script that needs to be transmitted to the sensor nodes. A small delta script translates to smaller reprogramming time and energy due to less number of packet transmissions over the network and less number of flash writes on the node. Figure 7 and Table 1 compare the delta script produced by Deluge, Stream, Rsync, semi-optimized Rsync, Optimized Rsync, and Zephyr. For Deluge and Stream, the size of the information to be transmitted is the size of the binary image while for the other schemes it is the size of the delta script. Deluge, Stream, Rsync, and semi optimized Rsync take up to 1987, 1324, 49, and 6 times more bytes than Zephyr, respectively. Note that for cases belonging to moderate or large change, the application

level modifications of Zephyr contribute to reducing the size of delta script significantly compared to optimized Rsync. Optimized Rsync takes up to 9 times more bytes than Zephyr. These cases correspond to shifts of some functions in the software. As a result, application-level modifications have great effect in those cases. In practice, these are probably the most frequently occurring categories of changes in the software. Case 1 and Case G are parameter change cases which do not shift any function. As a result, we find that delta scripts produced by optimized Rsync without application-level modifications are only slightly larger than the ones produced by Zephyr. Also even for very large software change cases (like cases 6, F, and C) Zephyr is more efficient compared to other schemes. In summary, application-level modifications have the greatest effects in moderate and large software change cases, significant effect in very large software change case (in terms of absolute delta size reduction) and small effect on very small software change cases.

Comparison with other incremental approaches: Rsync represents the algorithm used by Jeong and Culler [8] to generate the delta by comparing the two executables without any application-level modifications. We find that [8] produces up to 49 times larger delta script than Zephyr. Rsync also corresponds approximately to the system in [19] because it also compares the two executables without any application-level modifications. Koshy and Pandey [13] use a slop region after each function to minimize the likelihood of function shifts. Hence the delta script for their best case (i.e. when none of the functions expands beyond its slop region) will be same as that of Zephyr. But even in their best case scenario, the program memory is fragmented and less efficiently used than in Zephyr. This wastage of memory is not desirable for memory-constrained sensor nodes. When the functions *do* expand beyond the allocated slop region, they need to be relocated and as a result, all calls to those functions need to be patched with the new function addresses giving larger delta script than in Zephyr. Flexcup [13], though capable of incremental linking and loading on TinyOS, generates high traffic through the network due to large sizes of symbol and relocation tables. Also, Flexcup is implemented only on an emulator whereas Zephyr runs on the real sensor node hardware.

In the software change cases that we considered, the time to compile, link (with the application-level modifications) and generate the executable file was at most 2.85 seconds and the time to generate the delta script using optimized Rsync was at most 4.12 seconds on a 1.86 GHz Pentium processor. These times are negligible compared to the time to reprogram the network, for any but the smallest of networks. Further these times can be made smaller by using more powerful server-class ma-

chines. TinyOS applies extensive optimizations on the application binaries to run it efficiently on the resource-constrained sensor nodes. One of these optimizations involves inlining of several (small) functions. We do not change any of these optimizations. In systems which do not inline functions as TinyOS, Zephyr's advantage will be even greater since there will be more function calls. Zephyr's advantage will be minimum if the software change does not shift any function. For such a change, the advantage will be only due to the optimized Rsync algorithm. But such software changes are very rare, e.g. when only the values of the parameters in the program are changed. Any addition/deletion/modification of the source code in any function except the one which is placed at the end of the binary will cause the following functions to be shifted.

8.3 Testbed experiments

We perform testbed experiments using Mica2 [2] nodes for grid and linear topologies. For the grid network, the transmission range R_{tx} of a node is set such that $\sqrt{2}d < R_{tx} < 2d$, where d is the separation between the two adjacent nodes in any row or column of the grid. The linear networks have the nodes with R_{tx} such that $d < R_{tx} < 2d$, where d is the distance between the adjacent nodes. Due to fluctuations in transmission range, occasionally a non-adjacent node will receive a packet. In our experiments, if a node receives a packet from a non-adjacent node, it is dropped, to achieve a truly multi-hop network. A node situated at one corner of the grid or end of the line acts as the base node. We provide quantitative comparison of Zephyr with Deluge [7], Stream [16], Rsync [8] and optimized Rsync without application-level modifications. Note that Jeong and Culler [8] reprogram only nodes within one hop of the base node, but we used their approach on top of a multi hop reprogramming protocol to provide a fair comparison. The metrics for comparison are reprogramming time and energy. We perform these experiments for grids of size 2x2 to 4x4 and linear networks of size 2 to 10 nodes. We choose four software change cases, one from each equivalence class: Case 1 for Class 1 (SC), Case D for Class 2 (MC), Case 7 for Class 3 (LC), and Case C for Class 4 (VLC). Note that in the evaluations that follow, Rsync refers to the approach by Jeong and Culler [8].

8.3.1 Reprogramming time

Time to reprogram the network is the sum of the time to download the delta script and the time to rebuild the new image. Time to download the delta script is the time interval between the instant t_0 when the base node sends the first advertisement packet to the instant t_1 when the last node (the one which takes the longest time to download the delta script) completes downloading the delta script. Since clocks maintained by the nodes in the net-

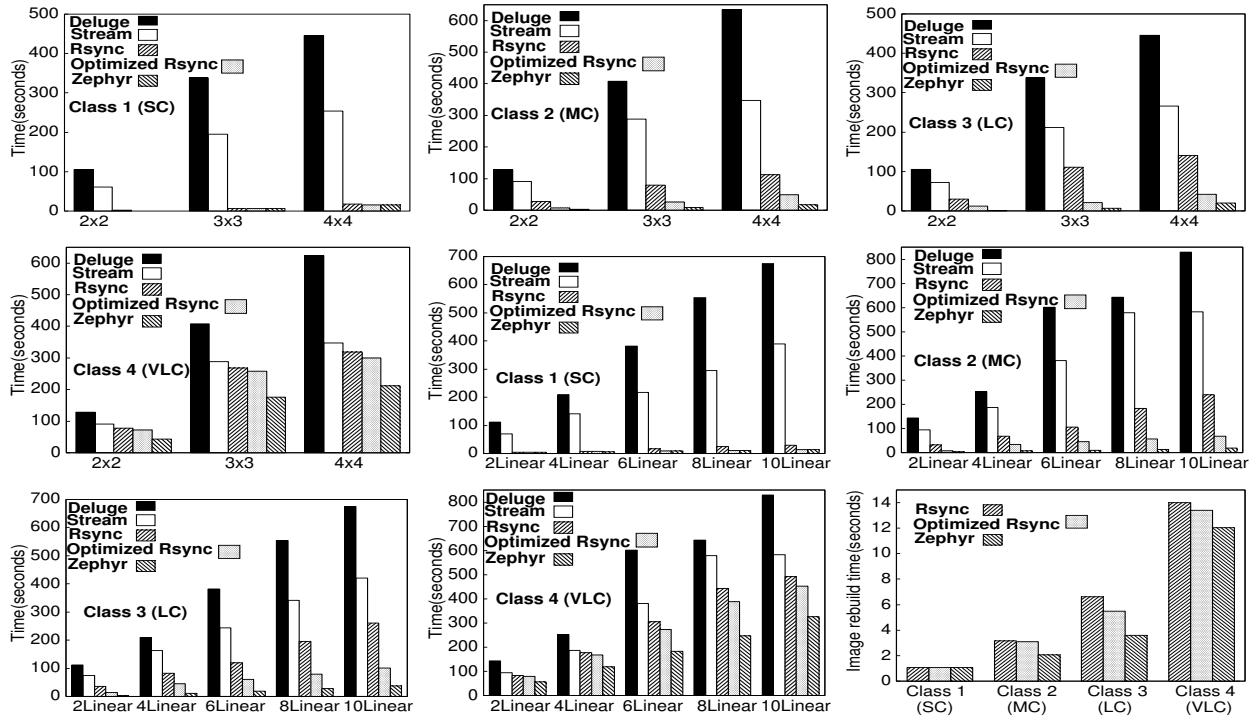


Figure 8: Comparison of reprogramming times for grid and linear networks. The last graph shows the time to rebuild the image on the sensor node.

Table 2: Ratio of reprogramming times of other approaches to Zephyr

	Class 1(SC)			Class 2(MC)			Class 3(LC)			Class 4(VLC)		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Deluge:Zephyr	22.39	48.9	32.25	25.04	48.7	30.79	14.89	33.24	17.42	1.92	3.08	2.1
Stream:Zephyr	14.06	27.84	22.13	16.77	40.1	22.92	10.26	20.86	10.88	1.54	2.23	1.46
Rsync:Zephyr	1.03	8.17	2.55	5.66	12.78	8.07	5.22	10.89	6.50	1.34	1.71	1.42
Optimized Rsync:Zephyr	1.01	1.1	1.03	2.01	4.09	2.71	2.05	3.55	2.54	1.27	1.55	1.35

work are not synchronized, we cannot take the difference between the time instant t_1 measured by the last node and t_0 measured by the base node. To solve this synchronization problem, we use the approach of [17], which achieves this with minimal overhead traffic.

Figure 8 (all except the last graph) compares reprogramming times of other approaches with Zephyr for different grid and linear networks. Table 2 compares the ratio of reprogramming times of other approaches to Zephyr. It shows minimum, maximum and average ratios over these grid and linear networks. As expected, Zephyr outperforms non-incremental reprogramming protocols like Deluge and Stream significantly for all the cases. Zephyr is also up to 12.78 times faster than Rsync, the approach by Jeong and Culler [8]. This illustrates that the Rsync optimization and the application-level modifications of Zephyr are important in reducing the time to reprogram the network. Zephyr is also significantly faster

than optimized Rsync without application-level modifications for moderate, large, and very large software changes. In these cases, the software change causes the function shifts. So, these results show that application level modifications greatly mitigate the effect of function shifts and reduces the reprogramming time significantly. For small change case where there are no function shifts, Zephyr, as expected, is only marginally faster than optimized Rsync without application-level modifications. In this case, the size of the delta script is very small (17 and 23 bytes for Zephyr and optimized Rsync respectively) and hence there is not much to improve upon. Since Zephyr transfers less information at each hop, Zephyr's advantage will increase with the size of the network. The last chart in Figure 8 shows the time to rebuild the new image on the sensor node. It increases with the increase in the scale of the software change, but is negligible compared to the total reprogramming time.

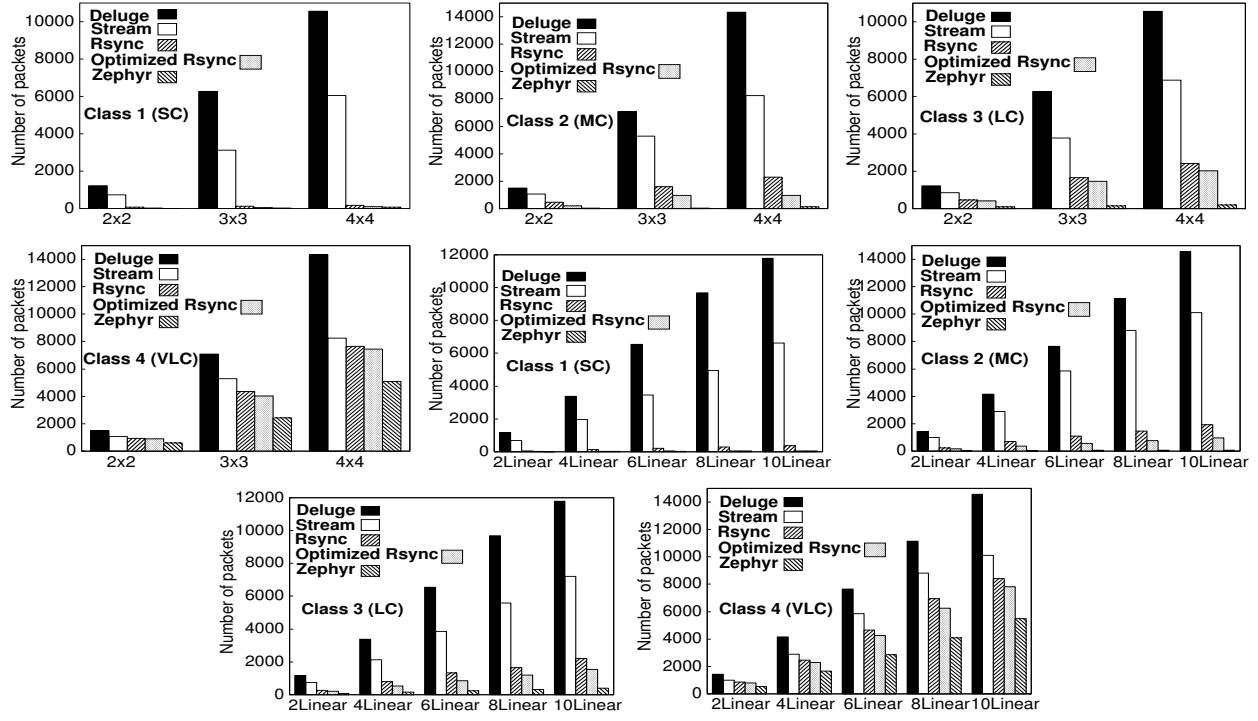


Figure 9: Comparison of number of packets transmitted during reprogramming.

Table 3: Ratio of number of packets transmitted during reprogramming by other approaches to Zephyr

	Class 1(SC)			Class 2(MC)			Class 3(LC)			Class 4(VLC)		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Deluge:Zephyr	90.01	215.39	162.56	40	204.3	101.12	12.27	55.46	25.65	2.51	2.9	2.35
Stream:Zephyr	53.76	117.92	74.63	28.16	146.1	82.57	8.6	36.19	15.97	1.62	2.17	1.7
Rsync:Zephyr	2.47	7.45	5.38	6.66	38.28	21.09	3.28	12.68	6.69	1.50	1.78	1.60
Optimized Rsync:Zephyr	1.13	1.69	1.3	4.38	22.97	9.47	2.72	10.58	3.95	1.38	1.64	1.49

8.3.2 Reprogramming energy

Among the various factors that contribute to the energy consumption during reprogramming, two important ones are the amount of radio transmissions and the number of flash writes (the downloaded delta script is written to the external flash). Since both of them are proportional to the number of packets transmitted in the network during reprogramming, we take the total number of packets transmitted by all nodes in the network as the measure of energy consumption. Figure 9 and Table 3 compare the number of packets transmitted by Zephyr with other schemes for grid and linear networks of different sizes. The number of bytes transmitted by all nodes in the network for reprogramming by Deluge, Stream, Rsync, and optimized Rsync is up to 215, 146, 38, and 22 times more than that by Zephyr. The fact that $Rsync:Zephyr > 1$ indicates that Zephyr is more energy efficient than the incremental reprogramming approach of [8]. The application-level modifications are

significant in reducing the number of packets transmitted by Zephyr compared to optimized Rsync without such modifications. Note that in cases like Case 7 and Case D (moderate to large change class), application-level modifications have the greatest impact where the functions get shifted. Application-level modifications preserve maximum similarity between the two images in such cases thereby reducing the reprogramming traffic overhead. In cases where only some parameters of the software change without shifting any function, the application-level modifications achieve smaller reduction. But the size of the delta is already very small and hence reprogramming is not resource intensive in these cases. Even for very large software changes, Zephyr significantly reduces the reprogramming traffic.

8.4 Simulation Results

We perform TOSSIM [12] simulations on grid networks of varying size (up to 14x14) to demonstrate the scalability of Zephyr and to compare it with other schemes.

Figure 10 shows the reprogramming time and number of packets transmitted during reprogramming for Case D (Class 2 (MC)). We find that Zephyr is up to 92.9, 73.4, 16.1, and 6.3 times faster than Deluge, Stream, Rsync [8], and optimized Rsync without application-level modifications, respectively. Also, Deluge, Stream, Rsync [8], and optimized Rsync transmit up to 146.4, 97.9, 16.2, and 6.4 times more number of packets than Zephyr, respectively. Most software changes in practice are likely to belong to this class (moderate change) where we see that application-level modifications significantly reduce the reprogramming overhead. Zephyr inherits its scalability property from Deluge since none of the changes in Zephyr (except the dynamic page size) affects the network or is driven by the size of the network. All application-level modifications are performed on the host computer and the image rebuilding on each node does not depend upon the number of nodes in the network.

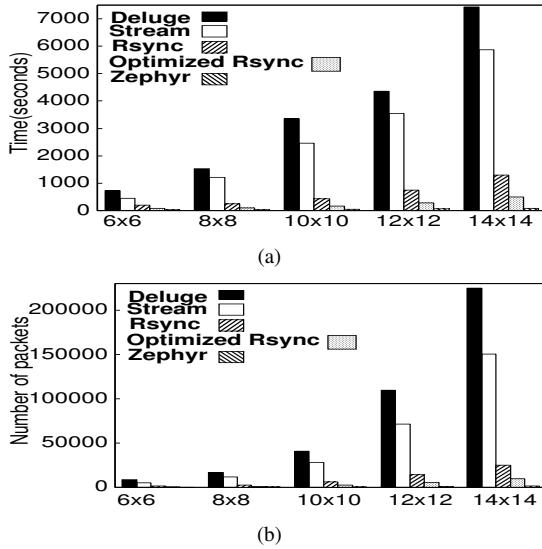


Figure 10: Simulation results for (a) reprogramming time and (b) number of packets transmitted during reprogramming (Case D, i.e. Class 2 (MC))

9 Conclusion

In this paper, we presented a multi-hop incremental reprogramming protocol called Zephyr that minimizes the reprogramming overhead by reducing the size of the delta script that needs to be disseminated through the network. To the best of our knowledge, we are the first to use techniques like function call indirections to mitigate the effect of function shifts for reprogramming of sensor networks. Our scheme can be applied to systems like TinyOS which do not provide dynamic linking on the nodes as well as to incrementally upload the changed modules in operating systems like SOS and Contiki that

provide the dynamic linking feature. Our experimental results show that for a large variety of software change cases, Zephyr significantly reduces the volume of traffic that needs to be disseminated through the network compared to the existing techniques. This leads to reductions in reprogramming time and energy. We can also use multiple nodes as the source of the new code instead of a single base node to further speed up reprogramming.

References

- [1] <http://www.tinyos.net>.
- [2] <http://www.xbow.com>.
- [3] <http://estadium.purdue.edu>.
- [4] <http://jarsync.sourceforge.net/>.
- [5] DUNKELS, A., GRONVALL, B., AND VOIGT, T. Contiki-a lightweight and flexible operating system for tiny networked sensors. *IEEE Emnets* (2004), 455–462.
- [6] HAN, C., RENGASWAMY, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. SOS: A dynamic operating system for sensor networks. *MobiSys* (2005), 163–176.
- [7] HUI, J., AND CULLER, D. The dynamic behavior of a data dissemination protocol for network programming at scale. *Sensys* (2004), 81–94.
- [8] JEONG, J., AND CULLER, D. Incremental network programming for wireless sensors. *IEEE SECON* (2004), 25–33.
- [9] KOSHY, J., AND PANDEY, R. Remote incremental linking for energy-efficient reprogramming of sensor networks. *EWSN* (2005), 354–365.
- [10] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review* (2002), 85–95.
- [11] LEVIS, P., GAY, D., AND CULLER, D. Active sensor networks. *NSDI* (2005).
- [12] LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. TOSSIM: accurate and scalable simulation of entire tinyOS applications. *Sensys* (2003), 126–137.
- [13] MARRON, P., GAUGER, M., LACHENMANN, A., MINDER, D., AND SAUKH, O., AND ROTHERMEL, K. FLEXCUP: A flexible and efficient code update mechanism for sensor networks. *EWSN* (2006), 212–227.
- [14] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. *SOSP* (2001), 174–187.
- [15] PANTA, R., AND BAGCHI, S. Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks. *To appear in IEEE Infocom* (2009).
- [16] PANTA, R., KHALIL, I., AND BAGCHI, S. Stream: Low Overhead Wireless Reprogramming for Sensor Networks. *IEEE Infocom* (2007), 928–936.
- [17] PANTA, R., KHALIL, I., BAGCHI, S., AND MONTESTRUQUE, L. Single versus Multi-hop Wireless Reprogramming in Sensor Networks. *TridentCom* (2008), 1–7.
- [18] PUCHA, H., ANDERSEN, D., AND KAMINSKY, M. Exploiting similarity for multi-source downloads using file handprints. *NSDI* (2007).
- [19] REIJERS, N., AND LANGENDOEN, K. Efficient code distribution in wireless sensor networks. *WSNA* (2003), 60–67.
- [20] TRIDGEELL, A. Efficient Algorithms for Sorting and Synchronization, PhD thesis, Australian National University.